Universidad
Carlos III de Madrid

Departamento de Tecnología Electrónica

PROYECTO FIN DE CARRERA

Implementación del algoritmo AES sobre arquitectura ARM con mejoras en rendimiento y seguridad.

Autor: Eduardo Bonilla Palencia

Tutor: Luis Mengibar Pozo

Madrid, 3 mayo de 2012

Título: Implementación del algoritmo AES sobre arquitectura ARM con mejoras en rendimiento y seguridad.

Autor: Eduardo Bonilla Palencia

Director: Luis Mengibar Pozo

EL TRIBUNAL

Presidente: Michael Victorio García Lorenz

Vocal: Francisco José Rodriguez Urbano

Secretario: Oscar Miguel Hurtado

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 11 de mayo de 2012 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

He de expresar mi profundo agradecimiento a todas aquellas personas que han contribuido al desarrollo de este proyecto. No puedo nombrar a todos, pero si quiero reconocer específicamente el valor a algunos de ellos.

A mi tutor D. Luis Mengibar Pozo por su constante ayuda y todas sus aportaciones tanto en el desarrollo del proyecto como en la redacción de esta memoria.

Gracias a mi familia por haber tenido la paciencia necesaria para que la carrera y este proyecto hayan llegado a su fin y por haber sido un pilar en el que apoyarme.

Me gustaría agradecer a todos los miembros del Casi Jugamos Bien su compañía y sus ratos de desahogo, gracias a ellos cualquier tarea es mucho más llevadera.

Por último agradecer a mis amigos y a todas aquellas personas que en algún momento han oído la frase “no puedo, tengo que ponerme con el proyecto”. Gracias a esos sacrificios este trabajo ha dado sus frutos.

Resumen

Este Proyecto Fin de Carrera se enmarca dentro de la implementación de algoritmos criptográficos para sistema empotrados seguros.

Concretamente, se ha desarrollado e implementado una versión mejorada del algoritmo AES en un microcontrolador con arquitectura ARM (ARM7 LPC 2132). Los microprocesadores con arquitectura ARM son ampliamente utilizados en dispositivos móviles tales como teléfonos inteligentes o tabletas, debido a su bajo consumo de potencia comparado con otras arquitecturas, y sus elevadas prestaciones. Tras un análisis detallado del algoritmo AES se han aplicado diversas técnicas para mejorar sus prestaciones y seguridad frente a ataques de canal lateral basados en trazas de consumo de potencia.

Con el objetivo de reducir el tiempo de procesamiento se han aplicado técnicas específicas para mejorar AES en arquitecturas de 32 bits. Aplicando estas técnicas se ha conseguido reducir el tiempo de procesamiento en un más de un 60% en la operación de cifrado y en un 90% en la operación de descifrado.

Aunque desde el punto de vista algorítmico AES es considerado como un cifrador seguro, se ha demostrado que no es inmune frente a algunos ataques basados el análisis de información obtenida de la propia implementación (ataques de canal lateral). Entre estos ataques se puede mencionar el análisis de las trazas de consumo de potencia o de las emisiones electromagnéticas del dispositivo electrónico que ejecuta el algoritmo. En este proyecto, además de las mejoras anteriormente citadas relativas al tiempo de procesamiento, se ha obtenido una implementación más robusta del algoritmo frente a ataques de consumo de potencia. Esto se ha conseguido mediante el enmascaramiento de la información procesada (masking), haciéndola de esta forma independiente del consumo.

Palabras clave: AES, cifrado, seguridad, máscaras, eficiencia, mejora, microprocesador, ARM7, ataques de consumo de potencia.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción y objetivos

1.1 *Introducción*

En la sociedad de hoy en día, las comunicaciones y el almacenamiento de información son digitales casi en su totalidad. Esto es una gran ventaja puesto que permite dinamizar y agilizar todas las operaciones pero también conlleva sus riesgos. Tanto los datos que enviamos por canales en abierto como los datos almacenados podrían ser leídos por cualquier persona con un mínimo de conocimientos si no los protegiéramos de ninguna manera.

En este marco, la ciencia de la criptografía tiene un papel crítico ya que nos permite mantener la privacidad.

De este modo, necesitamos privacidad para realizar operaciones que exigen una gran seguridad como pueden ser la banca electrónica, las compras electrónicas, el voto electrónico y demás aplicaciones.

Pero no solo utilizamos la criptografía para tareas tan cruciales, sin darnos cuenta a lo largo del día hacemos uso de ella en múltiples ocasiones: Cuando realizamos una llamada de teléfono móvil, cuando accedemos al correo electrónico o simplemente cuando accedemos a internet mediante una red WIFI.

Por las exigencias de estas aplicaciones, estos algoritmos deben ser robustos frente a ataques de cualquier tipo y a la vez rápidos para cumplir su objetivo y poderse usar en las tecnologías actuales, en muchos casos sensibles a retardos o con una capacidad de procesamiento limitada, especialmente en equipos portátiles o de bajo coste.

Un ejemplo de esto es el cifrado de GSM en el que la información de una ráfaga normal se codifica en dos paquetes de 58 bits cifrados, que son enviados por el terminal móvil cada 4.615 mseg [1]. Al tratarse de comunicaciones de voz, hay que evitar que este cálculo introduzca retardo alguno haciendo que sea lo menos pesado posible.

Así mismo, por seguir este ejemplo, también hay que tener en cuenta que en las comunicaciones GSM circula información cuya privacidad puede llegar a ser crítica, puesto que se pueden tener conversaciones con datos privados o incluso datos de pago por tarjeta de crédito. Por ello tampoco hay que olvidar su seguridad y habrá que reforzarla continuamente, adelantándose a las innovaciones en los ataques contra los algoritmos.

Por estos motivos vemos no solo que los algoritmos de cifrado tienen una importancia vital, sino que además deben adaptarse a las tecnologías actuales, minimizando los cálculos y maximizando la seguridad frente a ataques.

1.2 *Objetivos*

Al inicio del proyecto se definen un conjunto de subobjetivos para mejorar el diseño del algoritmo de cifrado, adaptándolo al entorno que usaremos, mejorando sus prestaciones e incrementando su seguridad protegiéndolo frente a ataques.

Implantación del algoritmo AES en el ARM 7 LPC2132.

El primero de los objetivos será implementar el algoritmo AES sin ningún tipo de mejora e integrarlo en el microprocesador LPC2132, tanto para el cifrado como para el descifrado y con distintos tamaños de clave.

Mejora de tiempos de funcionamiento aprovechando las características del microprocesador.

El objetivo es intentar reducir al máximo los tiempos de trabajo de varias maneras:

- En primer lugar adaptaremos el algoritmo a nuestro microprocesador, o sea que AES trabaje con palabras de 32 bits en vez de 8 bits.
- Además se modificará el código realizando en algunas funciones una serie de operaciones que reducirán un poco más este tiempo de cifrado.
- En último lugar trataremos de reducir más el tiempo de proceso utilizando técnicas de *loop unwinding*[1], o sea eliminando bucles de programación.

Protección frente a ataques.

Por último se aprovechará esa reducción de tiempo de proceso para robustecer el algoritmo protegiéndolo frente a ataques de canal lateral.

Para ello se perseguirán varias metas:

- Lo primero será obtener una semilla realmente aleatoria que nos permita obtener números indeterministas dentro del microprocesador.
- Se crearán máscaras aleatorias para ocultar tanto los datos comprometedores como las claves durante el cifrado o descifrado.
- Se buscará la mejor manera de integrar el algoritmo con las máscaras y de forma que se garantice la seguridad frente a ataques de canal lateral sin que repercuta en el rendimiento de la aplicación.

[1] Loop unwinding: También conocido como como loop unrolling, es una técnica de optimización que consiste en hacer más extenso un bucle simple reduciendo la sobrecarga por comparación del bucle y permitiendo una mejor concurrencia

Capítulo 2

Estado de la técnica

2.1 *Criptografía*

Según el Diccionario de la Real Academia Española [2], el significado de la palabra criptografía es "Arte de escribir con clave secreta o de un modo enigmático".

En un principio pudo llamarse arte, pero hoy en día es mucho más que eso, la *criptografía* es una tecnología que se basa en la Matemática Discreta, en la Teoría de la Información y en la Complejidad Algorítmica para dar seguridad al transporte y almacenamiento de la información.

Etimológicamente la palabra *criptografía* proviene de la unión de dos términos griegos: *κρυμμένο* (oculto) y *γραπτώς* (escritura), y es que se trata de escribir ocultando la información. Esta información se oculta escribiendo con una representación lingüística alterada del mensaje.

La criptografía sustituye la representación de la información, por otra representación, en principio inteligible, que es resultado de un algoritmo y una o varias claves de cifrado. Esta operación es reversible gracias a estas claves de cifrado.

Por tanto, el objetivo principal de la criptografía es el de mantener en secreto un mensaje. No obstante no es el único, puesto que la criptografía intenta además asegurar la autentificación de los datos (El emisor es quien dice ser), su integridad (Los datos no han sido modificados) y que el emisor no pueda negar haber enviado el mensaje.

2.1.1 Criptosistema

Un criptosistema es un conjunto formado por los posibles mensajes sin cifrar (M), los posibles mensajes cifrados (C), las posibles claves de cifrado (K), las transformaciones de cifrado (E) y las transformaciones de descifrado (D).

Todo criptosistema tiene que cumplir que si realizamos las transformaciones de descifrado con una clave a un mensaje cifrado con esa misma clave nos dé el propio mensaje como resultado:

$$D_k\big(E_k(m)\big) = m$$

2.1.2 Historia de la criptografía

El surgimiento de las computadoras ha marcado la historia de la criptografía separándola en dos partes claramente diferenciadas: La criptografía clásica y la criptografía moderna.

A continuación se relatan los principales sucesos dentro de estos dos periodos.

2.1.2.1 Criptografía clásica

Los secretos son parte de la vida del hombre prácticamente desde que éste existe, y con ellos la necesidad de conservarlos. Es por esto que el ser humano ha hecho uso de la criptografía desde mucho tiempo antes del que podamos imaginar.

En este apartado trataremos los inicios de la criptografía, desde sus albores hasta su modernización gracias a las computadoras.

La primera vez que se intentó ocultar información mediante cifrado fue entre el 600 y 500 a.c., cuando los hebreos hicieron uso de cifrado por sustitución monoalfabético, en el que todas las letras tenían otra correspondencia. La tabla de correspondencias sería la clave.

Más tarde, en la época de la Grecia clásica apareció el que se supone primer dispositivo físico de cifrado, la escitala. Los espartanos la usaban para realizar un cifrado por trasposición.

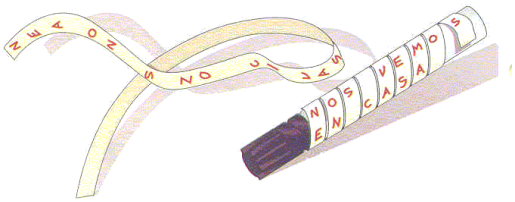

Figura 1. Escitala.

Su uso era muy sencillo, el mensaje estaba escrito cifrado en una tira de cuero u otro material. Al enrollarse alrededor de una escitala de determinadas dimensiones, los caracteres se trasponían y se podía leer el mensaje.

El primer algoritmo famoso que se conoce fue el algoritmo Cesar, que usaba Julio César para enviar mensajes secretos. Este algoritmo simplemente consistía en sumar 3 al número de orden de cada letra, por ejemplo en vez de escribir C se escribiría F.

Evidentemente estos tipos de cifrado por sustitución eran muy básicos, y realizando un análisis de frecuencia de aparición de caracteres se podían romper. Esta técnica, la del

análisis de frecuencias, fue inventada cerca del año 1000 y supuso la base sobre la que se asentaron los siguientes desarrollos en criptoanálisis hasta la segunda guerra mundial.

No fue hasta 1465 cuando se creó el cifrado por sustitución polialfabética, que era capaz de combatir el análisis de frecuencias gracias a que en este cifrado la sustitución aplicada a cada carácter varía en función de la posición que éste ocupe dentro del texto claro y de la clave. En práctica es una aplicación periódica de distintos cifrados monoalfabéticos.

Seguirían desarrollándose métodos de cifrado y de criptoanálisis pero no fue hasta la segunda guerra mundial cuando la criptografía sufrió la revolución que le haría llegar al nivel de desarrollo que tiene ahora gracias a las técnicas matemáticas.

2.1.2.2 Criptografía moderna

La criptografía moderna guarda una estrecha relación con las computadoras, de hecho se puede decir que nacieron al mismo tiempo. En la segunda guerra mundial, la función de *Colossus*, el que muchas personas llaman primer computador de la historia, era tratar de descifrar los mensajes enviados por el ejército alemán cifrados con la máquina *ENIGMA* o con el cifrado Lorenz.

Desde entonces, la tecnología de la criptografía ha tenido un increíble crecimiento. En un primer momento este crecimiento fue en secreto, por parte de agencias como la Agencia Nacional de Seguridad de los EE.UU. No obstante, en los últimos años las universidades de todo el mundo están logrando grandes avances en esta tecnología, haciéndolos públicos y permitiendo que tecnologías como la telefonía móvil, el comercio electrónico, o la distribución de contenidos multimedia puedan aprovecharse de ellos y ser mucho más fiables y seguras.

Este doble crecimiento, por un lado de una manera secreta y por el otro generalmente de manera pública provoca una carrera de descubrimientos, en la que el sector de las agencias secretas va por delante del ámbito académico pero éste sigue un buen ritmo y recopila todo el mérito de sus descubrimientos, aunque ya se hubieran descubierto en secreto el sector privado.

Así mismo, surge un dilema sobre la privacidad de los avances de la criptografía, en el que por un lado se defiende el no conocimiento de los algoritmos por la seguridad del cifrado mientras que por el otro lado se apoya publicar esos algoritmos para que sean mejorados por cualquier persona y la seguridad se base en la fortaleza del algoritmo.

Actualmente, después de analizar el dilema prácticamente todo el mundo defiende que resulta más conveniente que los algoritmos sean públicos. La fortaleza de un algoritmo no debe basarse en que sea secreto sino en la dificultad de atacarlo. Así toda la comunidad puede buscar puntos débiles y descubrir si un algoritmo es realmente bueno frente a ataques. Además, el hecho de que alguien pudiera acceder al código de un algoritmo privado pondría su seguridad en entredicho al basarse en impedir este acceso.

2.1.3 Fundamentos de la criptografía actual

Sea cual fuere la opción del tipo de algoritmo a desarrollar, la mayor parte de la teoría actual de la criptografía se apoya en dos trabajos:

- Los artículos "*A Mathematical Theory of Communication*" (1948) y "*Communication Theory of Secrecy Systems*" (1949) de Claude Shannon, en los

que aparecen las bases de la Teoría de la Información y de la Criptografía moderna. [3] y [4]

- "*New directions in Cryptography*" (1976), en el que Whitfield Diffie y Martin Hellman ampliaban la criptografía al campo de la criptografía Asimétrica. [5]

En estos dos trabajos aparecen por tanto las claves de la criptografía moderna y un nuevo tipo de criptografía que analizaremos a continuación.

2.1.3.1 Claves para ocultar la información

Shannon describió las dos claves para ocultar la información:

- Confusión. Oculta la relación entre el mensaje y el criptograma. La técnica más simple de confusión es la sustitución, que puede complicarse todo lo que queramos
- Difusión. Reparte toda la redundancia del texto claro a lo largo del texto cifrado. La técnica más sencilla de difusión es la transposición.

2.1.3.2 Criptografía simétrica y criptografía asimétrica

Gracias a las aportaciones de Diffie y Hellman los criptosistemas se dividieron en dos tipos en función del tipo de clave:

- C*riptosistemas simétricos* o *de clave privada*.

  En estos se emplea la misma clave para cifrar y para descifrar. Estos criptosistemas no permiten autentificación ni firma digital, solo confidencialidad.
  Son los que existieron desde un principio, y para las comunicaciones tienen el inconveniente de que deben conocer la clave únicamente el emisor y el receptor, por tanto la clave debe ser transmitida de una manera segura para que terceras personas no tuvieran acceso al contenido de los datos.
  Es por esto que conviene utilizarlos solamente para sistemas en los que la transmisión de la clave es segura o usar criptosistemas asimétricos para transmitir la clave.
  En el año 2002 comenzaron a comercializarse sistemas comerciales de basados en la criptografía cuántica.
  La criptografía cuántica aprovecha canales cuánticos para transmitir la clave. Este mecanismo se basa en que estos canales permiten que emisor y receptor detecten cuando un intruso accede al canal y por tanto pueden interrumpir la transmisión antes de comenzar a enviar el mensaje.
  Esto es consecuencia del principio de incertidumbre de Heisenberg, que dice que el proceso de medir en un sistema cuántico afecta al mismo sistema.
  En un principio usando estos canales para transmitir la clave privada se podrían utilizar criptosistemas simétricos con una gran confianza. No obstante se han realizado estudios [6] que prueban que los sistemas comerciales no son tan fiables como nos gustaría por lo que conviene insistir en mantener una gran seguridad en todos los eslabones de la cadena del cifrado.

- *Criptosistemas asimétricos* o *de clave pública*.

  Los criptosistemas asimétricos emplean una doble clave. Una clave privada (k_{pr}) y una pública (k_{Pb}). La clave pública y privada están relacionadas matemáticamente

pero esta relación es lo suficientemente compleja para que solamente el descifrador pueda averiguar la clave privada a partir de la pública.
Desafortunadamente, dicha complejidad provoca que las claves pública y privada sean creadas por un algoritmo que resulta costoso computacionalmente.
Por tanto estos sistemas son pesados para el codificador y decodificador por lo que en la práctica se usan para codificar la clave y enviarla con el mensaje codificado con un criptosistema simétrico al receptor.

2.1.4 Números aleatorios en criptografía

En criptografía los números aleatorios se usan principalmente de la mano de los algoritmos de clave pública para transmitir codificada la clave con que se codificará simétricamente el mensaje. Para añadir seguridad, esta clave deberá ser distinta e incorrelada cada vez y ahí es donde entran los números aleatorios.

Sin embargo, en este proyecto se trata solamente con algoritmos simétricos, por lo que solo se usarán los números aleatorios para obtener máscaras que den seguridad frente ataques al cifrado.

Pero, ¿Cómo se puede conseguir un número aleatorio?

Resulta difícil conseguir un número completamente aleatorio en un computador, de hecho con algoritmos deterministas no se puede conseguir más que secuencias pseudoaleatorias, que siguen un periodo de repetición. Las secuencias aleatorias se pueden clasificar en tres tipos:

- Secuencias estadísticamente aleatorias. Se trata de secuencias pseudoaleatorias con un periodo de repetición tan largo que superan los test estadísticos de aleatoriedad. A pesar de superar los test estas secuencias son completamente predecibles.
- Secuencias criptográficamente aleatorias. Son secuencias pseudoaleatorias impredecibles, de tal manera que computacionalmente resulta imposible averiguar el siguiente número de la secuencia.
- Secuencias totalmente aleatorias. Son secuencias que no pueden ser reproducidas de manera fiable. En ordenadores estas secuencias se pueden conseguir utilizando como semilla valores externos o independientes de la computadora. Estos valores pueden obtenerse de los dispositivos de entrada y salida de la computadora, del estado del registro de interrupciones o del reloj del sistema entre otras maneras.

2.1.5 Criptoanálisis (Tipos de ataques)

El criptoanálisis es el conjunto de técnicas que se pueden usar para romper los códigos criptográficos. Por tanto trata de comprometer la seguridad de un criptosistema, ya sea simétrico o antisimétrico.

No obstante, de esta definición debería excluirse averiguar el funcionamiento de un algoritmo secreto para aprovechar sus debilidades.

Aunque pueda pensarse que el criptoanálisis es el enemigo del criptosistema ya que trata de romperlo, la realidad es que ayuda a perfeccionar al criptosistema. Es aplicando técnicas de criptoanálisis como podemos ver cuáles son los puntos débiles de un algoritmo, perfeccionarlo y dificultar el posible criptoanálisis invasivo futuro.

El perfeccionamiento de los algoritmos ha provocado que para el criptoanálisis suela ser necesaria una computadora ya que en general se lleva a cabo analizando grandes cantidades de pares mensaje-criptograma.

Los principales tipos de análisis son:

- Texto claro escogido. Este tipo de ataque se basa en que conocemos algunos mensajes elegidos por nosotros y sus respectivos criptogramas. A partir de ahí se buscan relaciones para averiguar cómo sería el mensaje sin cifrar a partir del cifrado
- Fuerza bruta. El menos elaborado de todos, descifra el criptograma con todas las posibles contraseñas y de las posibles soluciones identifica las que tienen sentido.
- Análisis diferencial. Observa cómo afectan al criptograma ligeras modificaciones en el mensaje para deducir el criptograma a descifrar.
- Análisis lineal. Esta técnica consiste en realizar operaciones lógicas a pares mensaje-criptograma de las que se pueden sacar conclusiones sobre la clave de cifrado.
- Un tipo de ataque que solo se puede utilizar con los algoritmos asimétricos consiste en tratar de deducir la clave privada a partir de la pública.
- Ataques de canal lateral. Estudio de la potencia consumida por el componente electrónico que realiza el cifrado. Este método se basa en detectar qué operaciones está realizando el microprocesador a partir de su potencia consumida. De esta manera, conociendo el funcionamiento interno del algoritmo, si se observa la potencia que el sistema electrónico consume en el momento de operar con la clave, se puede identificar cual es la clave.

En este proyecto trabajaremos con el algoritmo AES. Está demostrado que este algoritmo es muy resistente a todos los ataques comentados anteriormente. De todas formas ante el de estudio de potencia consumida o de emisiones electromagnéticas no es tan férreo como frente al resto. Por tanto en la última parte de este proyecto implementaremos técnicas que hagan el algoritmo más robusto frente a este método de criptoanálisis.

2.1.6 Compromiso criptosistema criptoanálisis

Se puede elegir un algoritmo de cifrado tan complejo que sea prácticamente imposible atacarlo con cualquier técnica criptoanálisis conocida. No obstante esta complejidad tiene su coste (computacional, de almacenamiento e incluso económico) y podría llegar a hacer inservible el algoritmo.

Así mismo un algoritmo podría acarrear un coste ínfimo, pero lo más probable es que tuviera muy poca resistencia frente ataques de criptoanálisis.

Esto provoca que a la hora de elegir un sistema de cifrado tengamos que tomar un compromiso entre la resistencia al criptoanálisis y el coste del mismo.

En este proyecto usaremos el algoritmo AES en sus variantes de 128, 192 y 256 bits de longitud de clave por dos razones:

- Si realizáramos un ataque por fuerza bruta usando un computador que realizara un millón de operaciones por segundo contra un texto cifrado con una clave de

128 bits de longitud, tardaríamos más de 10^{24} años probar todas las posibles combinaciones. Es más si se diseñara una computadora lo suficiente rápida como para hacer esa cantidad de operaciones y su consumo de energía por operación fuera mínimo en cada cambio de estado, no habría suficiente energía en universo para que realizara todas las operaciones necesarias. Por tanto **se puede considerar que el algoritmo AES es suficientemente seguro aún con la menor longitud de clave.**

- Además, incluso funcionando con claves de 256 bits, el proceso de cifrado o descifrado de este algoritmo es altamente rápido comparado con el resto de algoritmos en distintas plataformas, tal y como se muestra en la figura 2.

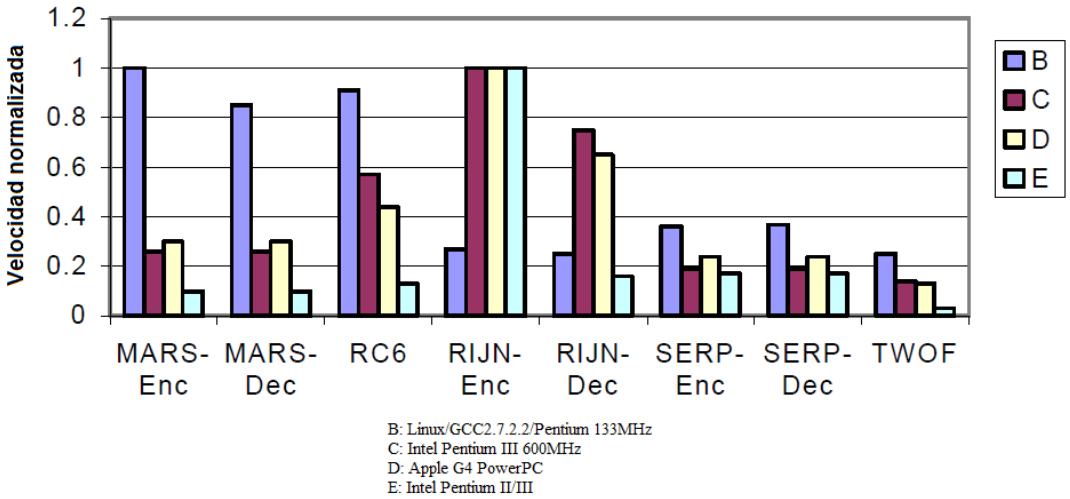

Figura 2. Comparativa de velocidades en distintos algoritmos. [7]

Debido a estas dos razones podemos ver por qué hemos elegido el algoritmo AES para trabajar. Además de que sea el estándar seleccionado por el NIST, vemos que su compromiso seguridad/coste es excelente.

2.2 *AES*

2.2.1 Orígenes. El algoritmo Rijndael

En 1993 los Belgas Joan Daemen y Vincent Rijmen, expertos en el campo de la criptografía, observaron la evolución del mundo que les rodeaba con importantes avances en el campo de las computadoras y encontraron una futura necesidad. Se dieron cuenta de que si estos avances seguían con su evolución lógica los algoritmos de cifrado de la época se quedarían obsoletos al cabo de unos pocos años y se pusieron a trabajar en un algoritmo de cifrado más robusto.

Sería en 1997 cuando publicaran los resultados. El algoritmo *Square* trabajaba con bloques de cifrado y claves del mismo tamaño, 128 bits. Esto suponía más del doble del tamaño de las claves de la mayoría de los algoritmos de la época.

El algoritmo de cifrado establecido como estándar por el Instituto Nacional de Estándares y Tecnología (NIST) era el *Data Encryption Standard* (DES), un algoritmo que para ser roto

por fuerza bruta exigía probar las 72.057.594.037.927.776 claves posibles. Este problema comenzaba a ser más que abordable por los ordenadores que venían apareciendo por lo que el algoritmo empezó a quedarse obsoleto.

Es por esto que en 1997 el NIST lanzó a concurso la selección de un nuevo sistema de cifrado, el *Advanced Encryption Standard* (AES). Durante el proceso de selección el NIST recomendó el uso del Triple DES como algoritmo de cifrado.

Este algoritmo debería ser simétrico, implementable tanto en hardware como en software y de dominio público. Así mismo debería poder usar claves de cifrado de 128, 192 y 256 bits, y usar bloques de cifrado de 128 bits como mínimo.

Estas especificaciones se acercaban mucho a las características del algoritmo *Square* por lo que en 1998 Daemen y Rijmen presentaron el algoritmo *Rijndael* (Combinación de los nombres de los autores) fruto de adaptar el algoritmo *Square* al concurso.

El día 20 de agosto de 1998 se habían presentado 15 candidatos al concurso:

Algoritmo	Autores	Puntuación
Cast-256	Entrust Technologies, Inc.	-2
Crypton	Future Systems, Inc.	-15
Deal	Richard Outerbridge, Lars Knudsen	-70
DFC	CNRS – Centre National pour la Recherche Scientifique – Ecole Normale Superieure	-5
E2	NTT – Nippon Telegraph and Telephone Corporation	14
Frog	TecApro International, S.A.	-85
HPC	Rich Schroeppel	-77
Loki97	Lawrie Brown, Josef Pieprzyk, Jennifer Seberry	-85
Magenta	Deutsche Telekom AG	-83
Mars	IBM	52
RC6	RSA Laboratories	73
Rijndael	John Daemen, Vincent Rijmen	76
Safer+	Cylink Corporation	-4
Serpent	Ross Anderson, Eli Biham, Lars Knudsen	45
Twofish	Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson	-85

Tabla 1. Candidatos para AES. [8]

Se creó un foro público para analizar los distintos algoritmos y el NIST designó un grupo de expertos para que valorara cuál era el oportuno en función de su robustez, estructura, capacidad de ser implementado tanto en software como en hardware o del trabajo intelectual realizado en el desarrollo.

Finalmente, en 1999 se seleccionaron estos cinco para una votación final.

Algoritmo	Votación final
Mars	13 votos
RC6	23 votos
Rijndael	86 votos
Serpent	59 votos
Twofish	31 votos

Tabla 2. Finalistas.

Curiosamente dos de los cinco finalistas estaban basados en la estructura del algoritmo *Square*, el *Rijndael* y el *Twofish*.

Una de las ventajas del algoritmo Rijndael frente al resto es que era el único que podía trabajar con claves y bloques de cifrado de 128, 192 y 256 bits indistintamente, pudiendo usarse 9 configuraciones distintas fruto de mezclar cualquier longitud de clave con cualquier longitud de bloque.

El algoritmo Rijndael se comporta muy bien tanto en software como en hardware, realizando un bajo uso de memoria y además tiene un tiempo de montaje de clave excelente. Además sus operaciones son fáciles de defender contra distintos tipos de ataque. Por tanto, finalmente el 2 de octubre de 2000, el NIST proclama al algoritmo Rijndael como ganador del concurso con este anuncio [9].

> When considered together, Rijndael's combination of security, performance, efficiency, ease of implementation and flexibility make it an appropriate selection for the AES.
>
> Specifically, Rijndael appears to be consistently a very good performer in both hardware and software across a wide range of computing environments regardless of its use in feedback or non-feedback modes. Its key setup time is excellent, and its key agility is good. Rijndael's very low memory requirements make it very well suited for restricted-space environments, in which it also demonstrates excellent performance. Rijndael's operations are among the easiest to defend against power and timing attacks.
>
> Additionally, it appears that some defense can be provided against such attacks without significantly impacting Rijndael's performance. Rijndael is designed with some flexibility in terms of block and key sizes, and the algorithm can accommodate alterations in the number of rounds, although these features would require further study and are not being considered at this time. Finally, Rijndael's internal round structure appears to have good potential to benefit from instruction-level parallelism.

En él se puede ver como su punto fuerte era la combinación de seguridad, rendimiento, eficiencia, flexibilidad y facilidad de implementación.

No obstante, al estandarizarse, el algoritmo AES no adopto todas las funcionalidades del Rijndael. El tamaño de bloque del AES quedó fijado únicamente en 128 bits, desechando la posibilidad de usar bloques de tamaño de 192 y 256.

Tras la selección, el algoritmo AES no solo sería usado para proteger la información del gobierno de los EE.UU. sino que lo usaría también el sector privado y se estandarizaría en muchos países (Sobre todo en los europeos).

A continuación intentaremos explicar cual es el funcionamiento interno que consiguió que el algoritmo Rijndael adquiriera estas características.

2.2.2 **Matemática del algoritmo**

Uno de nuestros objetivos es llegar a comprender el funcionamiento interno del algoritmo AES. No obstante, para lograrlo antes debemos comprender ciertos conceptos matemáticos en los que se basa este algoritmo como la aritmética modular.

La aritmética modular es una parte de las matemáticas muy útil en Criptografía, nos permite realizar cálculos complejos planteando problemas interesantes y manteniendo una representación numérica de los datos compacta y bien definida. Esto es posible gracias a que solo maneja un conjunto finito de números. Cuando el número representado supera un cierto valor llamado módulo, da la vuelta y vuelve a tomar el valor del primero de los números.

A nosotros nos interesa la aritmética modular puesto que el algoritmo Rijndael trabaja a nivel de byte, interpretando cada 8 bits como elementos de un campo de Galois GF (2^8).

Para poder entender el algoritmo desde todas sus facetas vamos a explicar en qué consiste un campo de Galois y como son sus operaciones

2.2.2.1 Campo de Galois

En el algoritmo Rijndael, todos los bytes se interpretan como elementos de un cuerpo finito, mediante Campos de Galois $\big(GF(K)\big)$.

Los campos de Galois tienen una gran importancia en criptografía porque representan un inverso aditivo y multiplicativo que permite cifrar y descifrar en el mismo cuerpo Z_K. Así eliminamos los posibles problemas de redondeo o truncamiento de valores que podrían surgir si usamos aritmética real.

En el algoritmo Rijndael interesa utilizar una aritmética en módulo *p* sobre polinomios de grado *m*, siendo p un número primo. Por tanto, este campo de Galois queda representado como

$$GF(p^m) = \{\lambda_0 + \lambda_1 x + \lambda_2 x^2 + \cdots + \lambda_{m-1} x^{m-1}; \lambda_0, \lambda_1, \lambda_2, \dots, \lambda_{m-1} \in Z_p\}$$

donde los elementos de $\mathrm{GF}(\mathrm{p}^{\mathrm{m}})$ se representan como polinomios de grado menor que *m* y con coeficientes Z_p.

Cada elemento de $\mathrm{GF}(\mathrm{p}^{\mathrm{m}})$ es un resto de módulo $p(x)$, donde $p(x)$ es un polinomio irreducible de grado *m*, es decir, no puede ser factorizado en polinomios de grado menor que *m*.

En el algoritmo Rijndael se utilizan los campos del tipo $\mathrm{GF}(2^{\mathrm{m}})$, puesto que los coeficientes en este caso serán los restos del módulo 2, es decir, 0 y 1, lo que permite una representación binaria. Por tanto, cada elemento del campo se representa con *m* bits y el número de elementos será 2^m.

Así por ejemplo, los elementos del campo $\mathrm{GF}(2^3)$ son: 0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1, que son los restos del polinomio de grado *m*-1.

Además se definen operaciones a nivel de bytes, por lo que nos basamos en un $\mathrm{GF}(2^8)$.

Este campo es un campo finito en el que los bytes serán representados como polinomios de grado 7 y con coeficientes binarios (0 o 1).

Un byte b en Rijndael se compone de 8 bits que se representan como $b_0, b_1, \dots, b_7$ donde b_7 es el bit de mayor peso y b_0 el de menor. Por tanto podemos representar el polinomio

como un polinomio cuyos coeficientes son los b_j con j = 0,1,…,7 y donde éstos b_j pueden valer 0 ó 1. Esta representación sería así:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i$$

A su vez se puede representar el valor de un byte usando notación hexadecimal de manera que cada mitad formada por 4 bits sea representada con un carácter.

Bits	Carácter	Bits	Carácter	Bits	Carácter	Bits	Carácter
0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

Tabla 3. Notación hexadecimal.

2.2.2.1.1 Suma en $GF(2^8)$

La suma de dos elementos del campo de Galois $\mathrm{GF}(2^8)$ es la suma de dos polinomios y por tanto da otro polinomio como resultado.

La peculiaridad es que en $\mathrm{GF}(2^8)$ las operaciones matemáticas sobre los coeficientes se hacen con módulo 2, por lo que en la suma si los coeficientes son iguales darán 0 como resultado y 1 si son distintos. Este funcionamiento es igual al de una OR-Exclusiva (XOR o $\oplus$) bit a bit.

La suma en el campo $\mathrm{GF}(2^8)$ cumple la propiedad asociativa, la conmutativa, tiene elemento neutro y tiene simétrico.

Así mismo, se define la operación resta de dos elementos como la suma de un elemento y el simétrico del otro.

2.2.2.1.2 Multiplicación en $GF(2^8)$

En el algoritmo Rijndael se emplea una multiplicación de dos elementos del conjunto $\mathrm{GF}(2^8)$. El resultado de esta multiplicación se expresa módulo $m(x)$ donde

$$m(x) = x^8 + x^4 + x^3 + x + 1 = \{11B\}$$

$m(x)$ es un polinomio irreducible ya que sus únicos divisores son el mismo y el 1.

En una multiplicación módulo $m(x)$ el resultado obtenido seguirá siendo un polinomio de grado menor que 8. De esta forma se asegura que la operación sigue siendo a nivel de bytes.

El proceso resulta bastante simple:

1. Se multiplican aritméticamente los polinomios.
2. Si aparecen valores de grado mayor a 8 se procederá a restar el polinomio $m(x)$ multiplicado por x^n con n tal que coincida con el valor de mayor grado. Se repetirá este paso hasta que quede un polinomio perteneciente a $\mathrm{GF}(2^8)$.

Este proceso se puede ver en el siguiente ejemplo en el que multiplicaremos los valores hexadecimales {57} y {83}:

$$r(x) = \{57\} * \{83\} = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

Anularemos el término x^{13} restando múltiplos de $m(x)$

$$m(x) * x^5 = x^{13} + x^9 + x^8 + x^6 + x^5$$

$$r(x) = r(x) - m(x) * x^5 = x^{11} + x^4 + x^3 + 1$$

Ahora se anulará el término x^{11} de la misma manera

$$m(x) * x^3 = x^{11} + x^7 + x^6 + x^4 + x^3$$

$$r(x) = r(x) - m(x) * x^3 = x^7 + x^6 + 1$$

De esta manera el resultado sigue siendo de grado menor que 8 por lo que pertenece a $\mathrm{GF}(2^8)$.

Esta operación es asociativa y distributiva con respecto a la suma y su elemento neutro es $\{01\}$.

También existe el polinomio inverso $a(x)$ a otro b(x) y cumplen

$$\big(a(x) * b(x)\big) \bmod m(x) = 1$$

2.2.2.1.3 Multiplicación por x en $GF(2^8)$

Un caso importante de la multiplicación de polinomios en $\mathrm{GF}(2^8)$ es la multiplicación de un polinomio por *x*. En este caso las potencias del polinomio se ven incrementadas en uno.

Si se multiplica un polinomio con grado menor que 7, este término ya pertenece a $\mathrm{GF}(2^8)$ por lo que no hará falta realizar más operaciones.

No obstante si el polinomio que se multiplica por *x* es de grado 7, aparecerá un término en grado 8, que desaparecerá dividiendo por el polinomio $m(x)$. Lo cual como hemos visto en el apartado anterior se resume restando por un término $m(x)$:

$$(b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) \bmod (x^8 + x^4 + x^3 + x + 1) =$$

$$(b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) - (x^8 + x^4 + x^3 + x + 1)$$

$$b_6x^7 + b_5x^6 + b_4x^5 + (b_3 - 1)x^4 + (b_2 - 1)x^3 + b_1x^2 + (b_0 - 1)x$$

Este cálculo se puede simplificar con cuatro funciones OR-Exclusivas. Tres sobre los bits b_3, b_2 y b_0 para la resta con los respectivos coeficientes de $m(x)$ y otra sobre el bit b_7 que compruebe su valor para saber si se tiene que realizar la función resta o la función identidad.

Para realizar este tipo de multiplicación, se define una función llamada $b = xtime(a)$, que simplifica la multiplicación de un polinomio por potencias de x.

Aplicando reiteradamente esta función podemos implementar de una manera sencilla y fácil el producto de un polinomio por una potencia de *x*:

$$a(x) * x = xtime\big(a(x)\big)$$

$$a(x) * x^2 = xtime\Big(xtime\big(a(x)\big)\Big)$$

$$a(x) * x^3 = xtime\Big(xtime\Big(xtime\big(a(x)\big)\Big)\Big)$$

…

Esta función resulta muy útil puesto que es la que se utilizará para calcular cualquier multiplicación de polinomios en $\mathrm{GF}(2^8)$ sumando cadenas de $xtime$. Por tanto, todas las multiplicaciones del algoritmo AES se pueden llevar a cabo con la función $xtime$. Su uso se centra en la función *MixColumns* del algoritmo con polinomios representados por $\{01\}$, $\{02\}$, $\{03\}$, $\{09\}$, $\{0B\}$, $\{0D\}$ y $\{0E\}$.

2.2.2.1.4 Representación de palabras en $GF(2^8)$

Una palabra es un conjunto de bytes. El algoritmo AES usa palabras de 4 bytes (32 bits), considerándolas como un polinomio de grado menor que cuatro con coeficientes que son a su vez polinomios en $\mathrm{GF}(2^8)$. De esta manera queda definida una palabra con esta expresión:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

La suma de palabras se realiza mediante operaciones OR-Exclusivas byte a byte, análogamente a lo realizado en $\mathrm{GF}(2^8)$. Esta operación no contrae problemas de desbordamiento puesto que como resultado siempre va a darnos otro polinomio de grado menor que 4.

En la multiplicación, al igual que en $\mathrm{GF}(2^8)$, si que nos podemos encontrar con el problema del desbordamiento. El producto de dos palabras de 4 bytes puede no ser representable por una palabra de 4 bytes, o sea, mediante un polinomio de grado menor que 4. Esta operación por tanto no es una operación interna en $\mathrm{GF}(2^8)$.

Por esto, se adopta la solución de expresar el resultado con módulo un polinomio de grado 4. Los autores del algoritmo eligieron el polinomio $M(x)$ como el módulo de la operación ya que todas las multiplicaciones que se pueden realizar en el algoritmo Rijndael se realizan con polinomios que poseen inverso.

$$M(x) = x^4 + 1$$

2.2.3 **Fundamentos de diseño del algoritmo Rijndael**

Joan Daemen y Vincent Rijmen diseñaron el algoritmo Rijndael basándose en tres criterios fundamentales:

- Resistencia contra la totalidad de los ataques conocidos.
- Velocidad, código compacto y operativo en multitud de plataformas distintas.
- Simplicidad de diseño.

Una de las principales diferencias entre Rijndael y el resto de los algoritmos de cifrado simétricos existentes radica en que Rijndael no tiene una estructura interna tipo *Feistel*. En una estructura tipo *Feistel*, en cada una de las operaciones de cada vuelta, todos los

bits sufren una permutación pero la mayoría de los bits no varían su valor, es decir, llegan a su nueva posición sin ningún cambio en su valor.

Rijndael no tiene una estructura tipo *Feistel* ya que trata todos los bits por vuelta. En Rijndael, cada vuelta está compuesta por tres transformaciones invertibles y distintas entre sí llamadas "layers" (capas). Estas transformaciones están basadas en el principio de diseño *"Wide Trail"* que da resistencia al algoritmo frente a ataques de tipo lineal y diferencial. Cada capa tiene su propia función específica dentro de esta estrategia:

- Capa de mezcla lineal: Garantiza una gran difusión de la información a través de la aplicación de varias rondas.
- Capa no lineal: Consiste en la aplicación paralela de la S-Box para conseguir unas óptimas propiedades de no linealidad
- Capa de adición de clave: Se mezcla el estado intermedio con la correspondiente subclave de ronda mediante una simple operación OR-Exclusiva.

Además, antes de la primera ronda del algoritmo, se aplica la capa de adición de clave. Esto se realiza para impedir que se ataquen directamente las claves. Por el contrario, cualquier capa que apliquemos tras la última adición de clave o antes de la primera, en los ataques basados en textos en claro conocidos, puede ser descubierta sin conocer la clave y esto sería un fallo en la seguridad del algoritmo, como se puede apreciar en la permutación inicial y final del algoritmo DES. Esta técnica no es nueva, puesto que ya aparece en algoritmos como IDEA o Blowfish.

En la última ronda, con el objetivo de que el algoritmo de cifrado y el de descifrado sean lo más similares posible, se aplica una capa de mezcla lineal distinta a la de las rondas anteriores. No obstante, esta técnica también aplicada en algoritmos como DES, no reduce la seguridad del algoritmo.

2.2.4 Funcionamiento del algoritmo AES

El algoritmo AES es un sistema de cifrado simétrico, es decir, utiliza la misma clave tanto para el cifrado como para el descifrado.

Además es un algoritmo de cifrado en bloque que trabaja con bloques de cifrado de 128 bits.

La clave puede ser de distintas longitudes, de 128, 192 o 256 bits, y en función de ella el algoritmo realizará un determinado número de rondas como queda reflejado en la tabla 4.

Longitud de clave	Número de rondas
128	10
192	12
256	14

Tabla 4.

Por tanto podemos decir que AES es un cifrador de bloque iterativo, que realiza varias rondas de cifrado sobre un bloque o matriz de 4x4 bytes llamada *estado* o *state*.

Tanto para el proceso de cifrado como para el de descifrado, en cada una de las rondas se utiliza una subclave generada anteriormente mediante la función KeyExpansion a partir de la clave de cifrado. La función KeyExpansion devuelve una subclave más que el número de rondas que tiene el algoritmo, esta subclave extra se utiliza antes de realizar todo el

proceso de cifrado o descifrado a la matriz de estado con el objetivo de imposibilitar el criptoanálisis.

En todas las rondas menos la última el algoritmo realiza una serie de operaciones que van modificando la matriz de estado aportando confusión o difusión al mensaje a cifrar.

A continuación se enuncian las operaciones que se aplican y posteriormente se describirán más detalladamente.

- Función SubBytes. Aporta confusión al proceso al realizar una sustitución byte a byte con propiedades óptimas de no linealidad.
- Función ShiftRows. Introduce difusión de información a la ronda mediante la rotación de las filas del estado.
- Función MixColumns. Permite un alto nivel de difusión mezclando las columnas entre sí.
- Función AddRoundKey. Introduce un grado de confusión que depende de la subclave de ronda.

La figura 3 es un simbólico resumen esquemático del funcionamiento de una ronda del proceso de cifrado

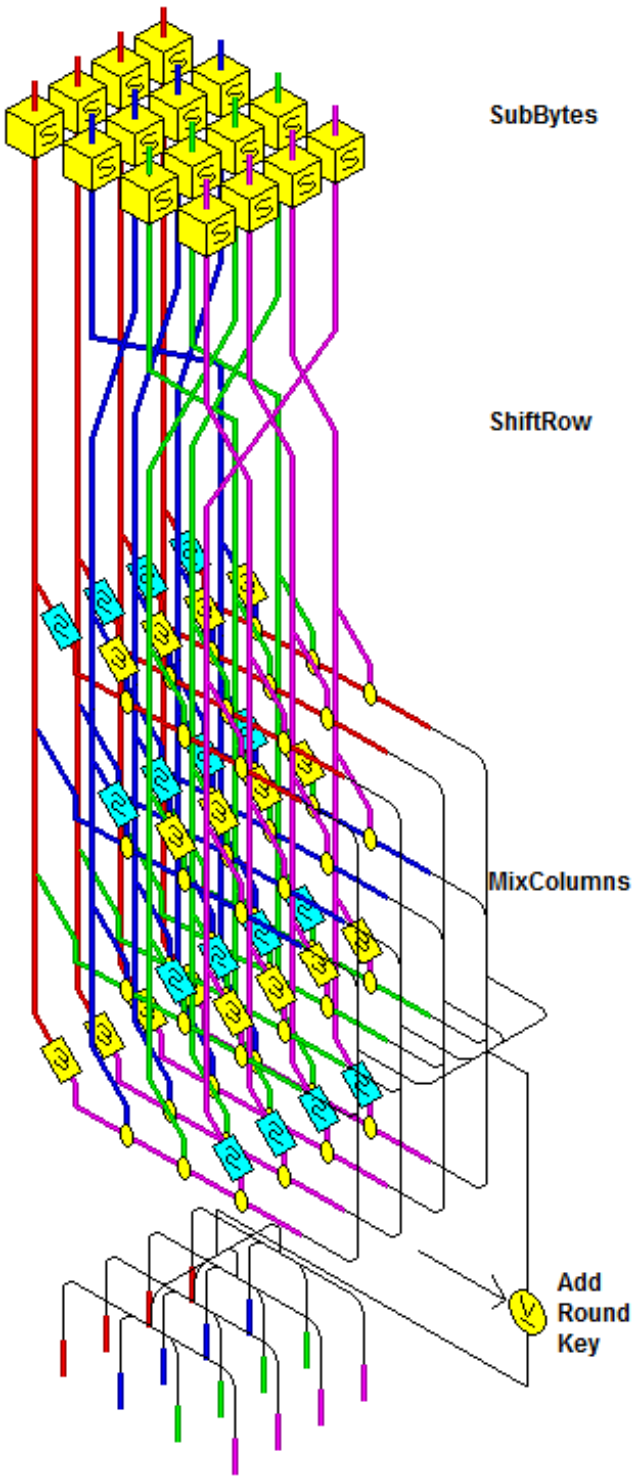

Figura 3. Resumen de una ronda de AES. [10]

2.2.4.1 Funcionamiento del proceso de cifrado

El proceso de cifrado consiste en aplicar cuatro funciones matemáticas invertibles a la información a cifrar. Estas funciones se repiten en cada vuelta.

La información a cifrar se va colocando en la matriz de estado, sobre la cual se realizarán las transformaciones.

Antes de realizar la primera ronda, se aplica la primera transformación a la matriz de estado. Se trata de la función AddRoundKey y consiste en una operación OR-Exclusiva entre la primera subclave y la matriz de Estado. A continuación, a la matriz de estado resultante se le aplican cuatro transformaciones invertibles (SubBytes, ShiftRows, MixColumns y AddRoundKey), repitiéndose estas cuatro transformaciones hasta llegar a la penúltima ronda, en lo que se conoce como la vuelta estándar.

Finalmente se le aplica una última ronda a la matriz de estado resultante de las anteriores vueltas, aplicando solamente las funciones SubBytes, ShiftRows y AddRoundKey en este orden. El resultado de la vuelta final produce el bloque cifrado deseado.

A continuación describiremos con más detenimiento las funciones que se realizan durante el proceso de cifrado. La referencia principal que se ha tomado tanto para las figuras como para la información escrita en los siguientes apartados ha sido la especificación de AES de la publicación FIPS correspondiente [11].

2.2.4.1.1 Función SubBytes (subBytes())

Tal y como se muestra en la figura 4, consiste en una sustitución no lineal que se realiza a nivel de bytes. Se aplica independientemente sobre todos y cada uno de los bytes que conforman la matriz de estado, generando una nueva matriz de bytes.

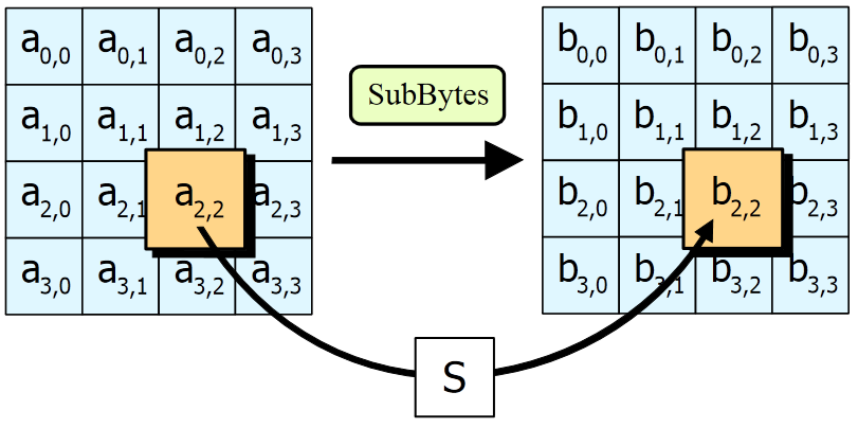

Figura 4. SubBytes.

De esta manera, se consigue aportar confusión al proceso para ocultar la relación entre el mensaje y el criptograma.

Esta transformación consiste en la sustitución de cada byte por el resultado de aplicarle la tabla de sustitución S-Box. La tabla S-Box es invertible y se construye mediante las siguientes dos transformaciones:

- Se sustituyen los elementos de la matriz de estado por sus inversos para la multiplicación en $\mathrm{GF}(2^8)$. Como el valor $\{00\}$ carece de inverso, en caso de aparecer, éste se sustituye por sí mismo.
- A continuación, se aplica la siguiente transformación afín en $\mathrm{GF}(2^8)$ al resultado de la anterior transformación.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

En esta transformación cada x_j representa el bit *j* del valor del byte resultante de la primera transformación y cuyo valor final se representa por y_j. Por tanto, cada y_j representa el bit *j* del valor del byte que constituye el resultado final de la transformación SubBytes.

Mediante estas dos transformaciones, aplicándolas a todos los posibles valores de entrada (256 valores ya que se trataba con un byte) se calcula una tabla de sustitución denominada S-Box que agiliza el proceso de cifrado.

Gracias a esta tabla, aplicar la función SubBytes resulta muy fácil. Consiste en dividir la matriz de estado en dos partes de cuatro bits. Los más significativos representados por *x,* con un rango de valores de 0 a 15, actúan de fila en la tabla y los menos significativos, representados por *y,* con el mismo rango de valores, actúan de columna en la tabla. El valor para esa fila y columna en la tabla es el resultado de aplicar S-Box a un byte.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 5. S-Box.

2.2.4.1.2 Función ShiftRows (shiftRows())

Esta transformación consiste en el desplazamiento de forma cíclica a la izquierda de los bytes de cada una de las filas de la matriz de estado.

Los bytes de la fila 0 de la matriz de estado no se desplazan, los de la fila uno se desplazan una posición a la izquierda, los de la fila 2 dos posiciones y los de la fila 3 se desplazan 3 posiciones a la izquierda. Por tanto, la transformación realizada sería la mostrada en la figura 6.

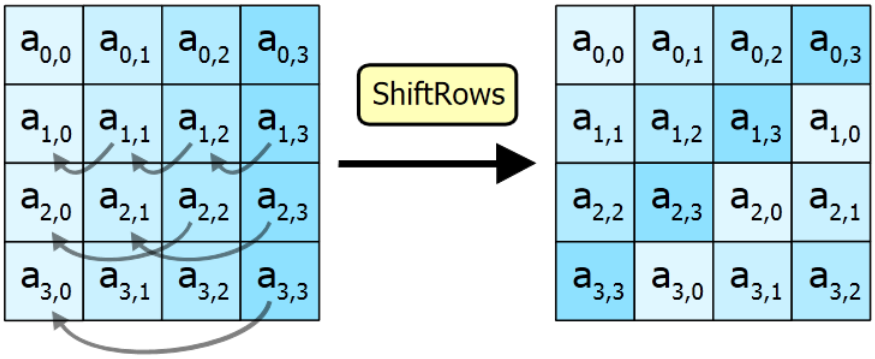

Figura 6. ShiftRows.

2.2.4.1.3 Función MixColumns (mixColumns())

La transformación MixColumns actúa sobre los bytes de una misma columna de la matriz de estado. Esta función permite una mezcla de los bytes de las columnas, tal y como se muestra en la figura 7.

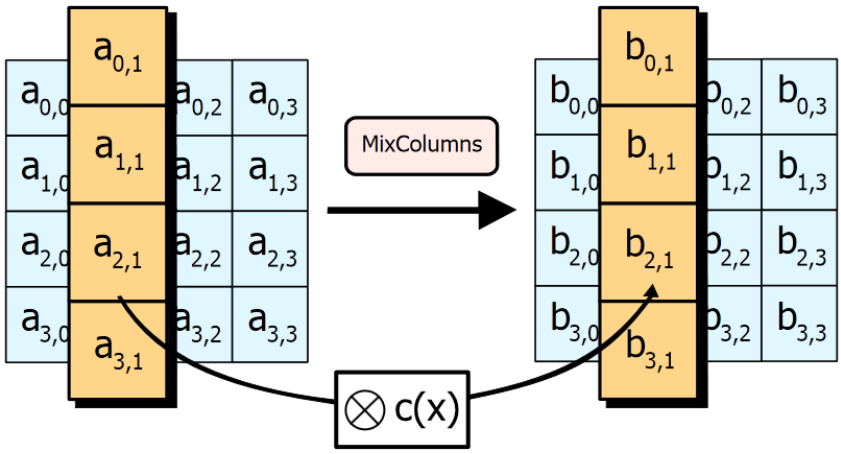

Figura 7. MixColumns.

En esta transformación las columnas de la matriz de estado, son consideradas polinomios con coeficientes del campo $\mathrm{GF}(2^8)$. Estos polinomios se multiplican módulo $M(x) = x^4 + 1$ por el polinomio $c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$

Matricialmente, esta transformación se podría resumir de la siguiente manera, representando los coeficientes b_i, las columnas *i* de la matriz de estado de salida y los coeficientes a_i las de la matriz de estado de entrada.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

2.2.4.1.4 Función AddRoundKey (addRoundKey(Subclave))

Esta operación consiste en una operación OR-Exclusiva entre los elementos de la matriz de estado que proviene de la transformación anterior y los elementos de la matriz de la subclave de ronda. Por tanto, la matriz de la subclave tiene las mismas dimensiones que la de estado, 4x4.

Esta función podría esquematizarse mediante la figura 8.

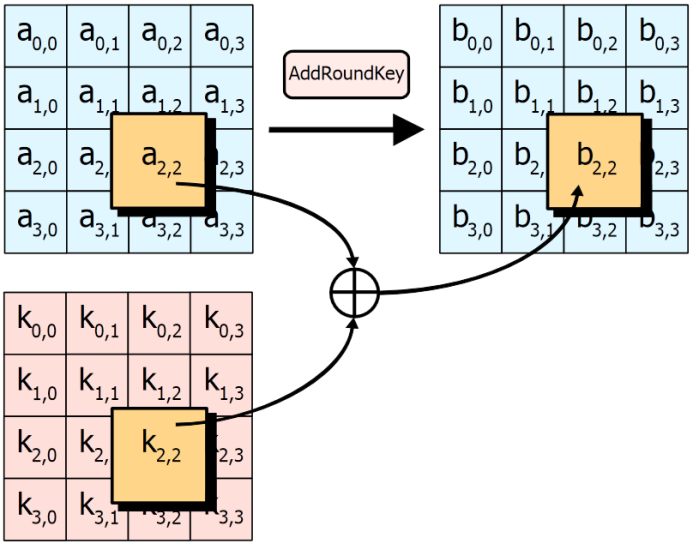

Figura 8. AddRoundKey.

El bloque resultante de esta función será la nueva matriz de estado para la siguiente ronda.

La esencia de esta función recae sobre las subclaves. El algoritmo AES utiliza diferentes subclaves k_j para que el resultado del algoritmo dependa completamente de una información externa al sistema: la clave de usuario. Así sigue el principio de la criptografía moderna, que establece que la seguridad de un algoritmo solo debe depender de la clave utilizada.

El funcionamiento de la transformación AddRoundKey es sencillo, lo verdaderamente interesante es conocer el procedimiento para generar las diferentes subclaves para cada ronda (RoundKey), subclaves que se derivan de la clave principal *k*. Para ello se utiliza la función KeyExpansion.

2.2.4.1.5 Función KeyExpansion (keyExpansion(clave))

Como hemos comentado anteriormente, la función KeyExpansion toma la clave de cifrado/descifrado y genera a partir de ella un vector de palabras de 4 bytes que de cuatro en cuatro formarán las distintas subclaves de ronda. El vector obtenido se denomina Clave Expandida.

La clave principal tiene N_k palabras de 32 bytes, pudiendo tomar N_k los valores 4, 6 y 8. Las primeras N_k palabras de la Clave Expandida son iguales a la clave de cifrado principal. El resto de palabras se obtienen recursivamente a partir de las primeras, dependiendo la función KeyExpansion de N_k.

Según describieron Daemen y Rijmen en su propuesta para AES, por motivos de seguridad hay dos posibles maneras de ejecutar la función KeyExpansion en función de N_k que ahora veremos en código C:

Para $N_k \leq 6$:

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
      for(i = 0; i < Nk; i++)
            W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);

      for(i = Nk; i < Nb * (Nr + 1); i++)
      {
            temp = W[i - 1];
            if (i % Nk == 0)
                  temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
            W[i] = W[i - Nk] ^ temp;
      }
}
```

Para $N_k > 6$:

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
      for(i = 0; i < Nk; i++)
            W[i] = (key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]);

      for(i = Nk; i < Nb * (Nr + 1); i++)
      {
            temp = W[i - 1];
            if (i % Nk == 0)
                  temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
            else if (i % Nk == 4)
                  temp = SubByte(temp);
            W[i] = W[i - Nk] ^ temp;
      }
}
```

En estas descripciones, la función SubByte(W) realiza una S-Box a cada uno de los bytes de una palabra de 32 bits y los devuelve en otra palabra.

La función RotByte(W) devuelve la palabra W rotada un byte a la izquierda.

Por tanto se puede ver como en ambos códigos las primeras N_k palabras de la Clave Expandida son las de la clave principal. Las siguientes palabras $W[i]$ son el resultado de una operación XOR entre la palabra anterior $W[i-1]$ y la palabra $W[i-N_k]$. Si la posición es múltiplo de N_k (En caso de que N_k sea mayor que 6, también si $i-4$ es

múltiplo de N_k), antes de la X-OR se tendría que realizar una transformación a $W[i-1]$ consistente en aplicar la función RotByte(W), la función SubByte(W) y una operación X-OR con una constante.

Las constantes aplicadas son generadas por la función Rcon de esta manera:

$$Rcon[i] = (RC[i], \{00\}, \{00\}, \{00\})$$

Siendo $RC[i]$ un elemento perteneciente al Campo de Galois $\mathrm{GF}(2^8)$ con valor de $x^{(i-1)}$ de manera que:

$$RC[1] = 1$$

$$RC[i] = x \cdot (RC[i-1]) = x^{(i-1)}$$

2.2.4.1.6 Pseudocódigo del proceso de cifrado

A continuación se añade un breve resumen en forma de pseudocódigo de lo comentado en este apartado.

```
State = BloqueACifrar;  //Introducimos el mensaje a cifrar

RoundKey[NumeroRondas + 1] = KeyExpansion(Clave);     //Generación de
                                                      //subclaves

AddRoundKey(State, RoundKey[0]);          //Primera subclave

for (i=0, i<NumeroRondas,i++)
      {
      //Rondas
      SubBytes(State);
      ShiftRows(State);
      MixColumns(State);
      AddRoundKey(State, RoundKey[i]);
      }

//Ronda final
SubBytes(State);
ShiftRows(State);
AddRoundKey(State, RoundKey[NumeroRondas]);

BloqueCifrado = State;  //Extraemos el mensaje cifrado
```

2.2.4.2 Funcionamiento del proceso de descifrado

Si se comprende el funcionamiento del proceso de cifrado, el de descifrado se puede resumir fácilmente. Consiste en sustituir las operaciones del proceso de cifrado por sus inversas y alterar el orden en que estas se aplican.

De esta manera el bloque a descifrar se colocaría en la matriz de estado y ésta sería sometida a una serie de transformaciones.

Antes de la primera ronda hay que aplicar la función AddRoundKey con la última subclave. A continuación se realizarían las rondas principales formadas por las operaciones InvShiftRows, InvSubBytes, AddRoundKey e InvMixColumns.

Finalmente habría que aplicar la ronda final, formada por las operaciones InfShiftRows, InvSubBytes y AddRoundKey. El resultado de esta última ronda produciría el bloque descifrado buscado.

Existen distintas variantes del orden de aplicación de estas operaciones y todas llegan al mismo resultado. Esto es debido a que por el funcionamiento interno de las mismas, se puede alterar el orden de las operaciones InvShiftRows e InvSubBytes por una parte y AddRoundKey e InvMixColumns por otra sin afectar al resultado final.

Estas operaciones son las del proceso de cifrado junto con una serie de cambios que explicaremos a continuación.

2.2.4.2.1 Función InvSubBytes (invSubBytes())

Para calcular la función inversa de SubBytes, bastará con aplicar una tabla inversa a la utilizada en el proceso de cifrado. Ésta se corresponde con la tabla 9.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 9. S-Box inversa.

2.2.4.2.2 Función InvShiftRows (invShiftRows())

Si para calcular la función ShiftRows había que desplazar los bytes de las filas 1,2 y 3 un total de 1, 2 y 3 posiciones a la izquierda, para la función inversa habrá que desplazarlas hacia la derecha

2.2.4.2.3 Función InvMixColumns (invMixColumns())

En la función InvMixColumns las columnas de la matriz de estado se consideraban polinomios y se multiplican módulo $M(x) = x^4 + 1$ por el polinomio $c(x) = 3x^3 + x^2 + x + 2$. En la función inversa bastará con hacerlo por el inverso del polinomio $c(x)$. El polinomio inverso ($a(x)$) se calcula mediante la definición de inverso:

$$a(x) \otimes c(x) = \{01\}$$

Y en este caso $a(x) = 11x^3 + 13x^2 + 9x + 14$

Por lo tanto, la transformación se podría resumir matricialmente de la siguiente manera.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 09 \\ 09 & 14 & 11 & 13 \\ 13 & 09 & 14 & 11 \\ 11 & 13 & 09 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Representando los coeficientes b_i, las columnas *i* de la matriz de estado de salida y los coeficientes a_i las de la matriz de estado de entrada.

2.2.4.2.4 Función AddRoundKey (addRoundKey(Subclave))

La función inversa de la operación AddRoundKey es la misma función AddRoundKey. Esto es debido a que AddRoundKey realiza una operación OR-Exclusiva entre los elementos de la matriz de estado y los de la matriz de la subclave, y el resultado de una operación XOR es invertible con la propia operación X-OR.

2.2.4.2.5 Función KeyExpansion (keyExpansion(clave))

Las distintas subclaves que se utilizan en el proceso de descifrado son las mismas que las del proceso de cifrado. La única diferencia que hay es que las distintas subclaves deberán tomarse en orden opuesto. Por lo tanto no es necesario modificar esta función, bastará con ir utilizando las subclaves en orden inverso como se puede ver en el pseudocódigo del apartado siguiente.

2.2.4.2.6 Pseudocódigo del proceso de descifrado

Este sería por tanto el pseudocódigo del proceso de descifrado.

```
State = BloqueADescifrar;     //Introducimos el mensaje a descifrar

RoundKey[NumeroRondas + 1] = KeyExpansion(Clave);     //Generación de
                                                      //subclaves

AddRoundKey(State, RoundKey[NumeroRondas]);     //Primera subclave

for (i=NumeroRondas-1, I>0,i--)
      {
      //Rondas
      InvShiftRows(State);
      InvSubBytes(State);
      AddRoundKey(State, RoundKey[i]);
      InvMixColumns(State);
      }

//Ronda final
InvShiftRows(State);
InvSubBytes(State);
```

2.2.5 **Modos de operación**

Hemos explicado el funcionamiento de un algoritmo de cifrado por bloques. Esto por si solo nos permite cifrar un mensaje del tamaño de un bloque de cifrado. Si queremos cifrar algo con mayor longitud deberíamos recurrir a los modos de operación.

Los modos de operación son una herramienta que nos permite usar repetidamente un algoritmo de cifrado con una única clave para cifrar de manera segura mensajes de longitud variable.

Antes de encriptar un mensaje tendríamos que dividir los datos en distintos bloques de cifrado y probablemente rellenar el último bloque mediante *padding*[2]. Posteriormente, se seguirán las directrices del modo de funcionamiento para cifrar todos los bloques por separado.

Cada modo de funcionamiento tiene diferentes directrices que aportan mayor o menor seguridad, eficacia o eficiencia.

Los modos principales modos de funcionamiento que se pueden usar junto con AES son ECB, CBC, OFB, CFB y CTR. Aparte de estos existen otros modos de funcionamiento pero como su importancia no es tan grande o son más específicos en cuanto a su objetivo, a continuación solo explicaremos los primeros.

[2] Padding: En criptografía y en el resto de ámbitos, los mensajes tienen una longitud indeterminada. Sin embargo hay muchas aplicaciones que tratan esos mensajes dividiéndolos en distintos bloques de igual longitud. Por tanto, el último bloque no siempre se rellenará completamente. Las técnicas de *padding* se encargan de rellenar de múltiples maneras este último bloque para que se pueda realizar la tarea asociada.

2.2.5.1 Electronic codebook (ECB)

Es el más sencillo de todos los modos de funcionamiento. Simplemente divide el mensaje en los distintos bloques y codifica por separado a todos con la misma clave.

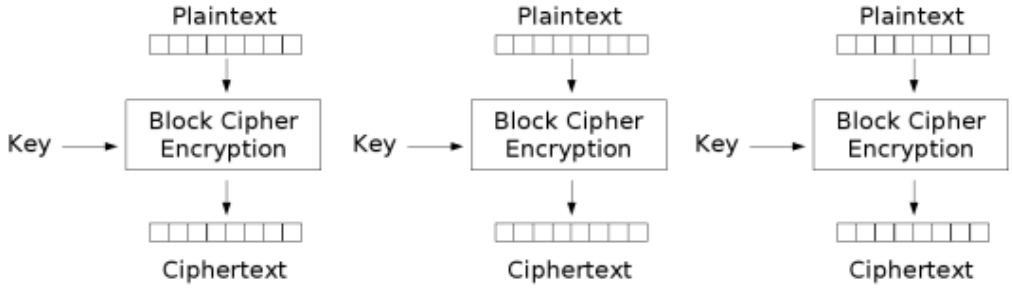

Figura 10. ECB.

La debilidad de este modo es que codifica bloques iguales de la misma manera, por lo que no esconde correctamente los patrones de los datos. Esto se puede apreciar perfectamente en la figura 11 en la que aparece una imagen excesivamente poco variable sin cifrar, cifrada con ECB y cifrada con CBC.

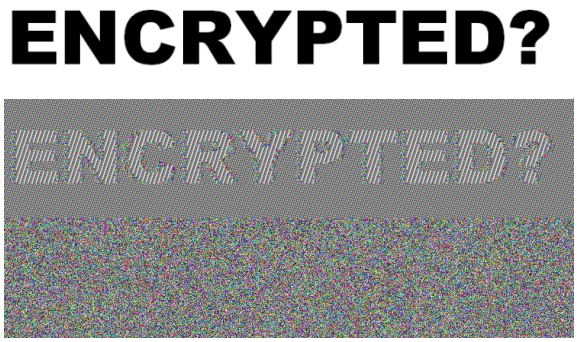

Figura 11. Debilidad ECB.

2.2.5.2 Cypher-block chaining (CBC)

En el modo CBC antes de cifrar cada uno de los bloques, se les aplica una operación XOR con el bloque cifrado anterior.

Además, si se pretende que este mensaje cifrado sea único, se le puede aplicar una operación XOR al primer bloque junto con un vector de iniciación que debería conocer quién vaya a decodificar el mensaje para recuperar el primer bloque.

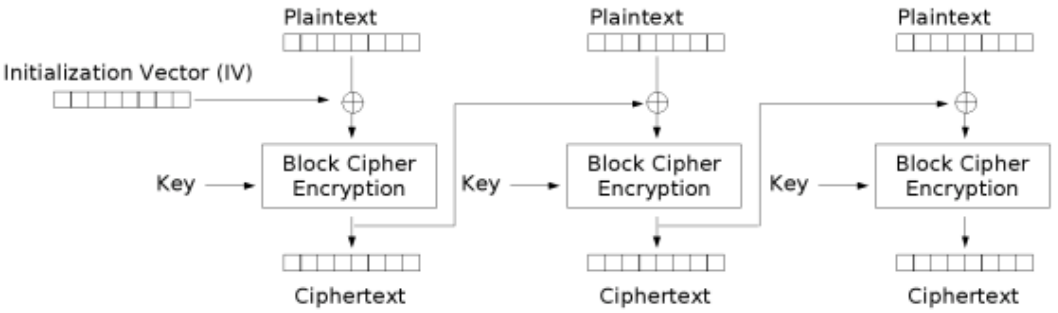

Figura 12. CBC.

Este modo de funcionamiento se usa con mucha frecuencia, aunque tiene el inconveniente de que al depender el cifrado de un bloque del anterior bloque cifrado, no se puede trabajar en paralelo con distintos bloques.

Además, si hubiera un fallo en el proceso, este sería propagado dos bloques más.

2.2.5.3 Output feedback (OFB)

El modo de funcionamiento OFB convierte el cifrado en bloque en un cifrado de flujo.

En él no se codifica directamente con AES el mensaje, si no que se parte de un vector de inicialización que es codificado mediante AES y el vector cifrado se suma al primer bloque a codificar mediante una operación XOR. El resto de bloques se codifican de la misma manera solo que cifrando en bloque la salida cifrada del anterior codificador de bloque en lugar del vector de inicialización.

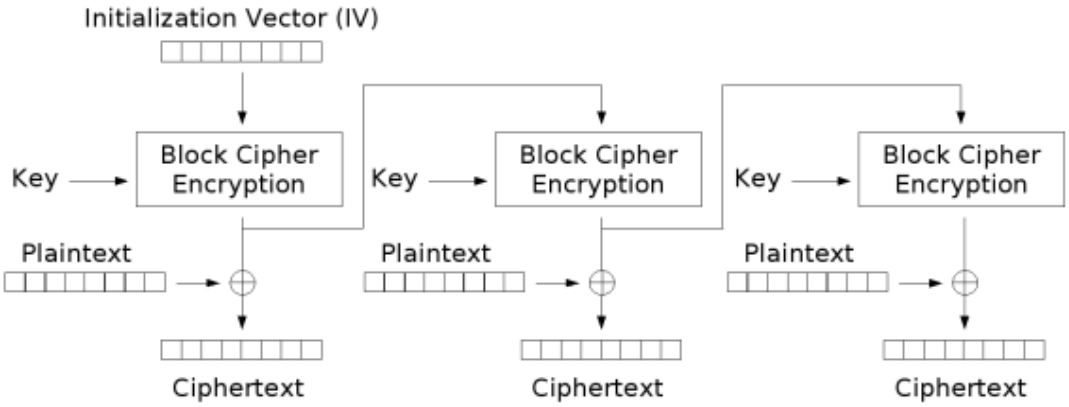

Figura 13. OFB.

Con el método OFB la parte más costosa tampoco podría realizarse en paralelo aunque una vez realizados todos los cálculos del algoritmo de bloque, si se podría realizar en paralelo el paso del texto origen al texto cifrado. Un fallo en el cifrado en bloque sería arrastrado, aunque no ocurriría lo mismo si este fallo ocurriera en la suma.

2.2.5.4 Cipher feedback (CFB)

El modo CFB comienza codificando un el texto cifrado del bloque anterior (O un vector de inicialización si fuera el primer bloque) y a la salida se le suma mediante una XOR el texto a codificar. El resultado será el bloque codificado.

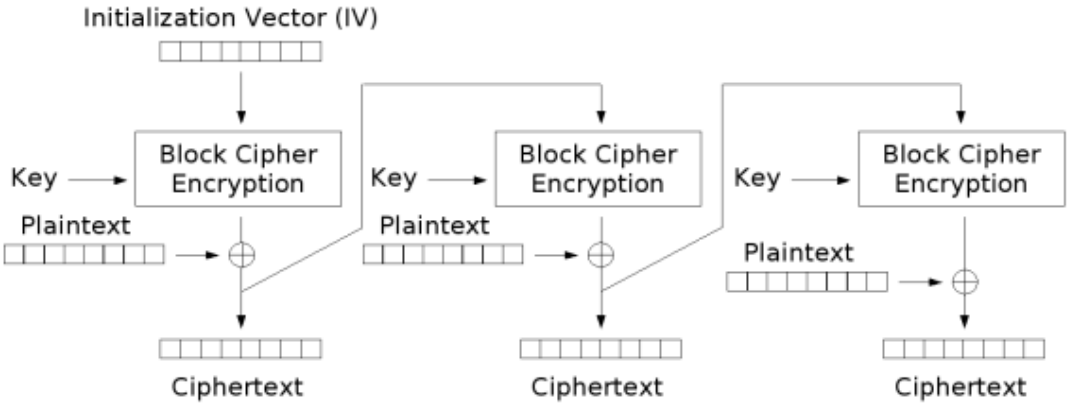

Figura 14. FCB.

CFB no puede realizarse trabajando en paralelo y los fallos también son arrastrados.

2.2.5.5 Counter (CTR)

En el modo CTR se parte de una secuencia o un contador cuyos valores son codificados mediante el algoritmo de bloque. Estos valores codificados se suman a los bloques de texto a cifrar mediante una operación XOR.

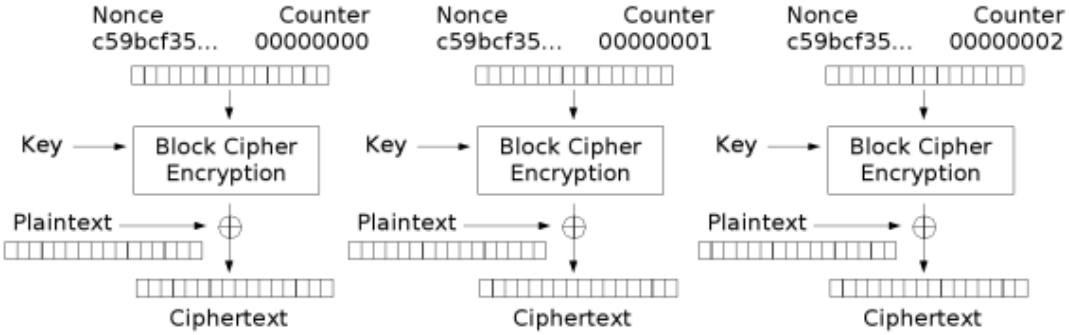

Figura 15. CTR.

De esta manera los mismos textos en distintos bloques tendrían distinta codificación. Además este modo de operación permite trabajar con cada bloque en paralelo reduciendo tiempos de cómputo y los errores no serían propagados.

CTR es un método ampliamente aceptado. No obstante, la secuencia que se usa tiene que ser cuidadosamente elegida puesto que puede ser una debilidad. No es bueno que sea una secuencia simple ni que tenga un ciclo de repetición corto.

Capítulo 3

Entorno de trabajo

3.1 *El microprocesador*

Ahora que conocemos el funcionamiento del algoritmo AES, tenemos que pensar en donde realizar su implementación. Atendiendo a la complicación de las operaciones de AES, se puede realizar sobre un computador de propósito general, un *ASIC*[3], una *FPGA*[4] o un microcontrolador.

Atendiendo a su uso, una unidad de cifrado y descifrado debe estar preparada para producirse a gran escala, en aplicaciones fijas o portátiles y deberá convivir con otras aplicaciones y procesos paralelos ya que siempre será utilizada por otras unidades.

Por ello de entre las opciones que se barajaban para implementarlo se ha considerado que la óptima para nuestra aplicación es un microcontrolador puesto que es un sistema de bajo coste que se puede utilizar para aplicaciones portátiles y puede alojar múltiples aplicaciones.

Un microprocesador es un circuito integrado que incorpora una unidad central de proceso (CPU) y todo un conjunto de elementos lógicos que permiten enlazar otros dispositivos a través de los buses formando un sistema completo para cumplir con una determinada aplicación tal y como se muestra en la figura 16. [12]

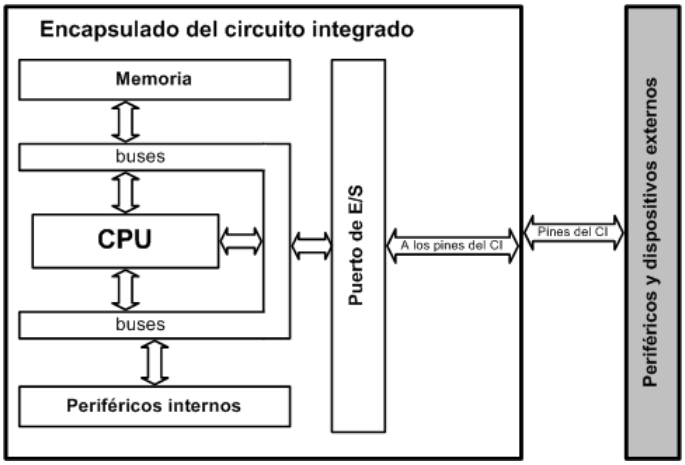

Figura 16. Estructura de un microprocesador.

Para que el sistema pueda realizar su labor debe ejecutar paso a paso un programa que consiste en una secuencia de instrucciones, almacenándolas en uno o más elementos de memoria.

[3] ASIC: Un Circuito Integrado para Aplicaciones Específicas (o ASIC, por sus siglas en inglés) es un circuito integrado hecho a la medida para un uso en particular, en vez de ser concebido para propósitos de uso general. Se usan para una función específica.

[4] FPGA: Una FPGA (del inglés Field Programmable Gate Array) es un dispositivo que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada rápidamente mediante un lenguaje de descripción especializado. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip. Las FPGAs se utilizan en aplicaciones similares a los ASICs no obstante son más lentas, tienen mayor consumo de potencia y no pueden abarcar sistemas tan complejos como ellos.

El microprocesador o simplemente procesador, es el circuito integrado más importante, de tal modo, que se le considera el cerebro de una computadora. Puede definirse como chip, un tipo de componente electrónico en cuyo interior existen miles o en ocasiones millones, según su complejidad, de transistores cuyas interacciones le permiten al chip realizar las funciones que se le han encargado.

Se dice que es el cerebro de la computadora porque es la parte que lleva a cabo y ejecuta los programas. Acomete instrucciones que se le dan a la computadora a muy bajo nivel realizando operaciones lógicas simples, como sumar, restar, multiplicar o dividir que uniéndose forman operaciones complejas. Tiene un lugar específico dentro de la placa, el cual suele estar refrigerado.

Su velocidad de proceso se mide en función de la cantidad de operaciones por ciclo que es capaz de realizar y los ciclos por segundo que ejecuta. Esta velocidad se puede indicar en Hertzios, refiriéndose a la frecuencia de reloj.

Más tarde realizaremos una descripción algo más detallada del microprocesador que se ha utilizado en este proyecto.

3.1.1 Elección del microprocesador utilizado

A la hora de elegir el microprocesador a utilizar desde un primer momento se tenía claro que este debía ser de la familia ARM.

Se denomina ARM (Advanced RISC Machines) a una familia de microprocesadores diseñados por la empresa Acorn Computers y desarrollados por Advanced RISC Machines Ltd.

Los microprocesadores ARM7 son unos microprocesadores con arquitectura de 32 bits, modernos y de bajo coste orientados a sistemas empotrados que ofrecen una muy buena capacidad de cómputo.

Las anteriores características fueron muy importantes a la hora de seleccionarlo para este proyecto. Sin embargo, la propiedad determinante fue el hecho de que los microprocesadores de la familia ARM son dominantes en el mercado de dispositivos móviles, llegando a utilizarse en millones de unidades de teléfonos móviles, tabletas o sistemas de videojuegos portátiles entre otras aplicaciones. En todas ellas es necesaria una capacidad de procesamiento relativamente elevada y un consumo de energía bajo.

En concreto el chip utilizado está presente en dispositivos tales como Game Boy Advance, Nintendo DS, Apple iPod, Lego NXT, y Juice Box entre otros.

A pesar de que el sector de los microprocesadores se renueva a una gran velocidad, el mercado sigue utilizando microprocesadores sucesores del ARM7TDMI, lo cual permite que la investigación realizada pueda ser aplicable en múltiples plataformas.

3.1.2 **ARM7TDMI-S LPC2132 FBD64**

Teniendo claro que el microprocesador tenía que ser un ARM7TDMI, se utilizó el microcontrolador ARM7TDMI-S LPC2132 junto con el JTAG y los cables de alimentación y de conexión entre microprocesador y JTAG.

Figura 17. Placa de desarrollo del LPC2132.

Según el fabricante [13] [14] [15], el microcontrolador incorpora:

- 64 kB de memoria flash ROM.
- 16 kB de memoria RAM.
- Mecanismo de emulación por JTAG, ISP ICE-RT.
- Controlador de interrupciones vectorizadas de prioridad configurable (VIC).
- 9 entradas de IRQ externa con disparo por nivel o flanco.
- 2 puertos serie asíncronos (UART0 y UART1).
- Puertos serie síncronos (2 Fast I2C y 2 SPI).
- 3 temporizadores de 32 bits.
- Un conversor analógico a digital.
- Un conversor digital a analógico de 8 canales de 10 bits.
- 46 pines de entrada/salida tolerantes a 3.3V
- 2 modos de bajo consumo.

En él los periféricos serán vistos por la unidad central (CPU) como registros que podrían ser:

- De datos. Contienen los datos que se van a usar en el periférico y que se comunicarán con la CPU.
- De estado. Dan información sobre en que estado se encuentra el periférico.
- De control. Se escriben para configurar el periférico

La mayor parte de estos registros son de 32 bits y para acceder a ellos se utiliza el mapeado en memoria, se accede a ellos como si se accediera a una dirección de memoria.

Además cuenta con un subsistema de reloj que puede dar una frecuencia de funcionamiento de 60 MHz.

La figura 18 muestra un esquema de la placa de desarrollo del procesador. En ella figura como se comunica el procesador con el JTAG, los distintos pines del microprocesador y sus posibles usos y otras opciones de configuración.

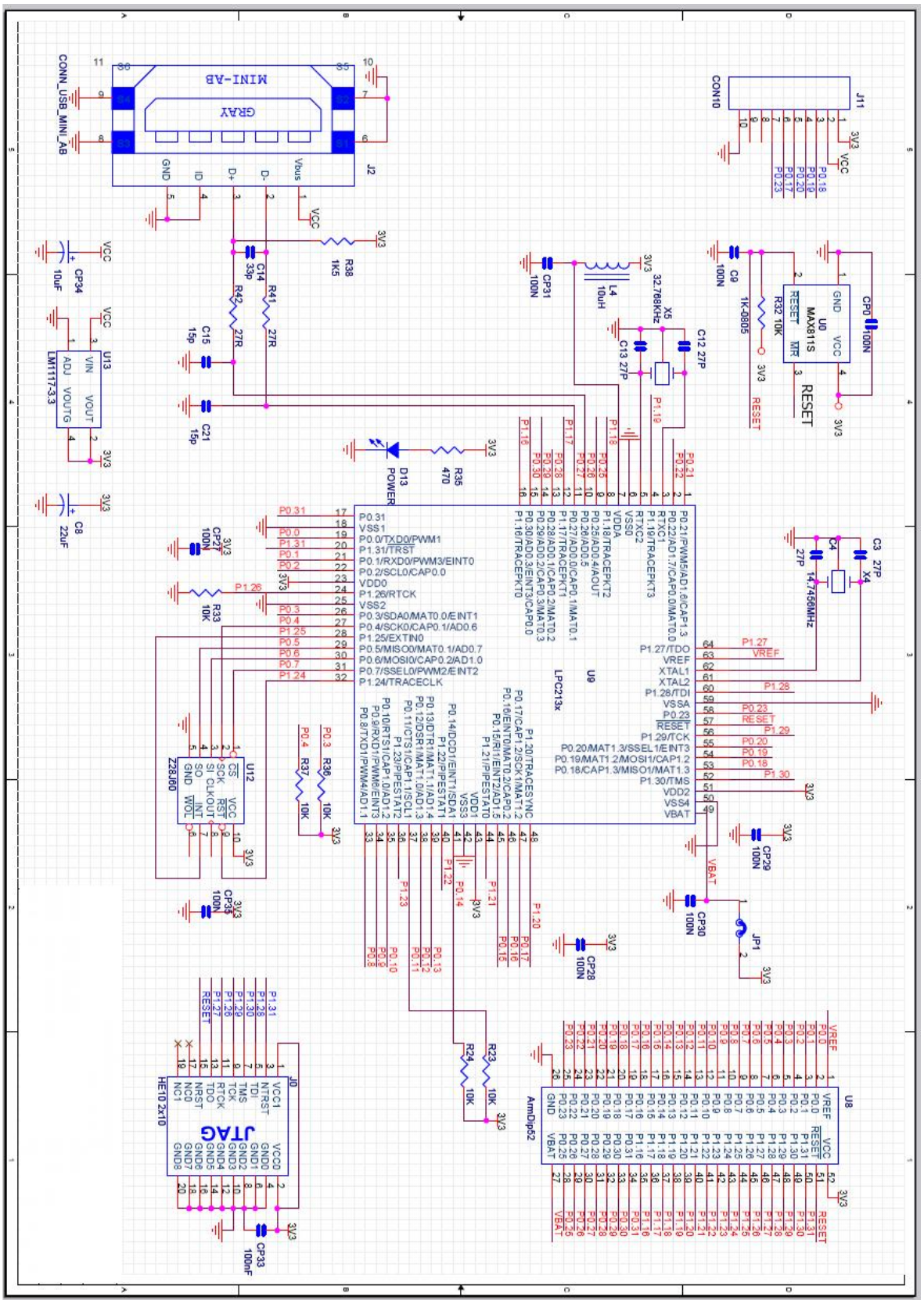

Figura 18. Esquema de la placa de desarrollo del LPC2132.

El *JTAG*, o *ULINK*, es el dispositivo que se utiliza para cargar los programas en el microprocesador así como para realizar emulaciones en el propio procesador.

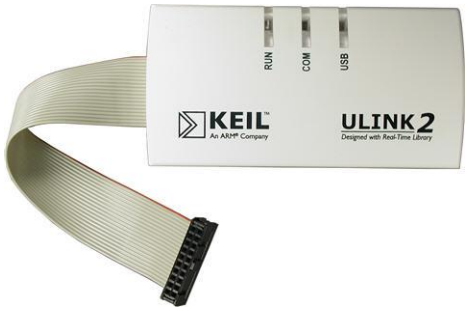

Figura 19. ULINK.

3.2 *Desarrollo del programa. Keil uVision4*

El software que implementa el algoritmo AES ha sido desarrollado mediante el programa uVision4 del fabricante Keil.

uVision es una herramienta específicamente desarrollada para microprocesadores como el ARM7 que nos permite crear programas en código C o ensamblador para el microcontrolador, compilarlos, emularlos y simularlos o ejecutarlos sobre el propio microcontrolador.

Además incorpora la funcionalidad de mostrar en una simulación o emulación el tiempo que consume el microprocesador en un conjunto de operaciones. Realizando una serie de medidas se ha comprobado que esta herramienta es del todo fiable, por lo que podemos tomar medidas de tiempo directamente de este programa.

3.3 *Lenguaje de programación C*

Como ya hemos comentado, uVision4 permite programar el microprocesador mediante código C o ensamblador.

En este proyecto se ha decidido usar el lenguaje de programación C debido a que al no ser tan específico como el ensamblador puede reutilizarse en muchas otras plataformas.

C es un lenguaje de programación no orientado a objetos simple y flexible. Utiliza tipos primitivos para evitar operaciones sin sentido y punteros para referenciar posiciones de memoria. A pesar de ser un lenguaje no orientado a objetos, permite el encapsulado y polimorfismo de una manera rudimentaria gracias a los punteros a funciones y variables estáticas.

3.4 *Cable de comunicación serie – USB (TTL-232R-3v3)*

Atendiendo a las características de nuestro microprocesador, se decidió transmitir los datos a cifrar/descifrar y los datos cifrados/descifrados entre el ordenador y el microchip a través del puerto serie.

El puerto serie es un sistema de comunicaciones perfectamente adaptado al LPC2132, sin embargo, en los ordenadores actuales cada vez se usa más el puerto USB en lugar del

serie. Es por esto que se decidió utilizar el cable de comunicaciones serie-USB TTL-232R-3V3.

Figura 20. Cable de comunicación serie – USB.

Para más información sobre el cable TTL-232R-3v3 se insta a consultar la bibliografía [16].

3.5 *Ordenador personal de 32 bits con Windows XP*

El programa de desarrollo (Keil uVision4) y otras utilidades necesarias para el desarrollo de este proyecto exigían el uso de un ordenador personal.

Por tanto se decidió utilizar un ordenador con procesador de 32 bits y Windows XP como sistema operativo.

3.6 *HyperTerminal*

HyperTerminal es un programa de comunicaciones que se incorporó en los sistemas operativos de Microsoft entre Windows 95 y Windows XP. Se encuentra en el menú de inicio, Programas, Accesorios, HyperTerminal.

Permite establecer una comunicación Serie entre el ordenador y el microprocesador.

Capítulo 4

Implementación y mejoras del algoritmo

En este capítulo se describe el diseño que se ha realizado e implantado del algoritmo AES así como las mejoras de rendimiento y de seguridad que se han aplicado.

El capítulo se divide en tres apartados, en los que se explican los principales bloques del proyecto:

1. Implementación de AES en el microcontrolador.
2. Mejoras de rendimiento.
3. Mejoras de seguridad.

Cada uno de estos bloques se corresponde con un diseño del algoritmo que variará respecto al diseño anterior gracias una serie de mejoras desarrolladas en el mismo apartado.

Para la comprobación del correcto funcionamiento de los programas realizados se han utilizado los vectores de prueba propuestos por el NIST en la descripción de los modos de funcionamiento para algoritmos de cifrado en bloque [17].

Resumiendo, el algoritmo AES se ha implementado para operar con todos los posibles tamaños de clave (128, 192 y 256 bits), tanto para cifrar como para descifrar. Primero siguiendo las especificaciones del NIST. A continuación aplicando mejoras de rendimiento para procesadores de 32 bits. Por último, aplicando mejoras de seguridad frente a

ataques de canal lateral. Esto se ha hecho en lenguaje de programación C para su implementación en el microprocesador LPC2132 mediante el programa Keil uVision4.

4.1 *Implementación de AES en el LPC2132*

En este primer apartado no se buscaba aplicar ninguna mejora de funcionamiento al algoritmo AES. Se ha buscado implantar el cifrador y descifrador de AES para todos los posibles tamaños de clave en el microprocesador y que éste sea plenamente funcional. Esta implementación respeta la especificación del NIST.

4.1.1 **Funcionamiento del programa**

A continuación se muestra la figura 21 con el diagrama de flujo del programa que más tarde se ampliará dividiéndolo en dos partes para explicarlo mejor.

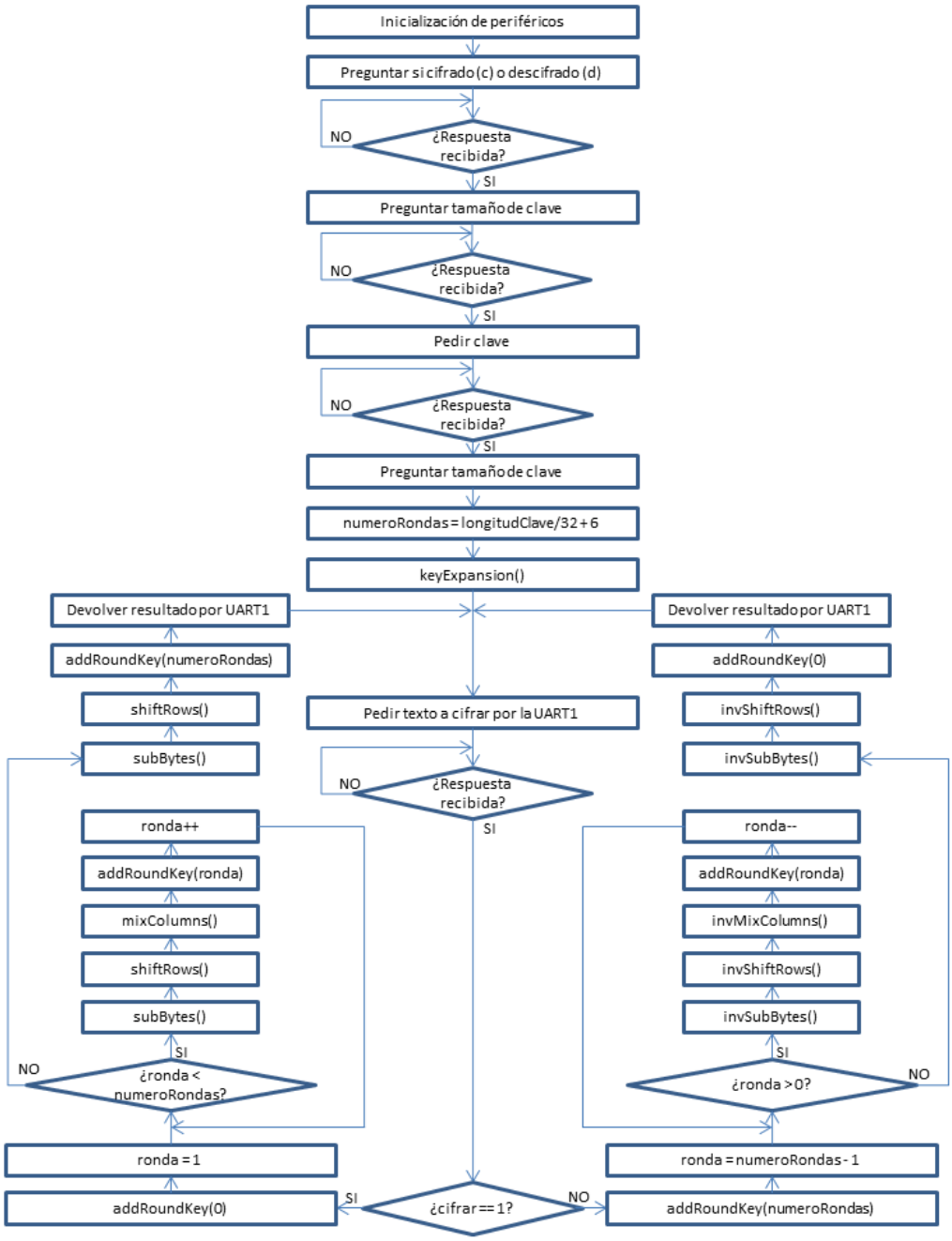

Figura 21. Diagrama de flujo del programa.

El programa diseñado es capaz de ejecutar el algoritmo AES tanto para cifrar como para descifrar con tamaños de clave de 128, 192 y 256 bits.

Con el objetivo de que la aplicación sea lo más eficiente posible, dentro de la inicialización de periféricos se configura además la frecuencia de trabajo del microprocesador y de los periféricos, y se establece en 60Mhz, que es la mayor frecuencia posible. De este modo la ejecución se realizará a la máxima velocidad.

La manera de comunicarse que utiliza es el puerto serie UART1 del microprocesador. Por este puerto se transmiten los parámetros de cifrado así como el tamaño de clave. Estos parámetros son: el modo de cifrado/descifrado, la clave de cifrado y los datos de entrada y de salida. Además se transmiten indicaciones para que la interacción con el microprocesador sea la correcta.

Por tanto habrá una primera fase que llamaremos de *toma de parámetros* en la que el programa inicializará la mayor parte de las herramientas del microprocesador y establecerá las características de la operación.

La figura 22 se corresponde con un diagrama de flujo más específico que seguirá el programa durante la fase de toma de parámetros.

Si nos fijamos en el diagrama podremos observar que en la toma de parámetros la correspondencia entre los caracteres recibidos y el modo de funcionamiento utilizado es la representada en la tabla siguiente:

Carácter recibido	Modo de funcionamiento
'd' o 'D'	Descifrado
Cualquier otro	Cifrado
'3'	Tamaño de clave de 256 bits
'2'	Tamaño de clave de 192 bits
Cualquier otro	Tamaño de clave de 128 bits

Tabla 5. Correspondencia de parámetros de inicialización.

De esta manera escriba lo que escriba el usuario habrá una acción asignada y se evitarán errores de funcionamiento.

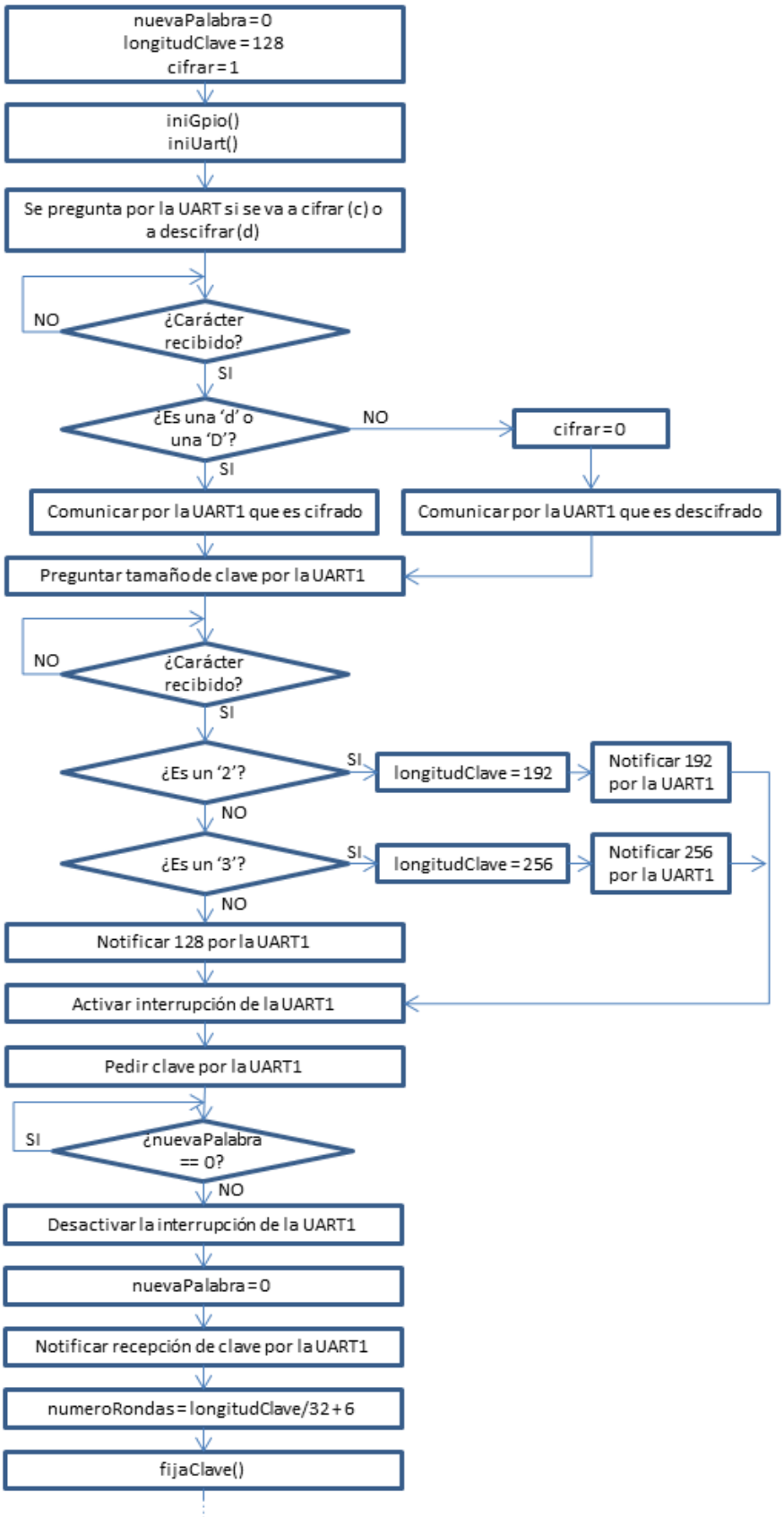

Figura 22. Fase de toma de parámetros.

A continuación se desarrolla una segunda fase que podremos llamar de *ejecución de AES* en la que el programa permanecerá hasta que se reinicie.

En esta fase el programa generará las subclaves de ronda y cíclicamente esperará un bloque de cifrado, le aplicará las operaciones de cifrado o descifrado, y devolverá por la el puerto serie los caracteres resultantes del proceso.

Esta fase queda representada en la figura 23.

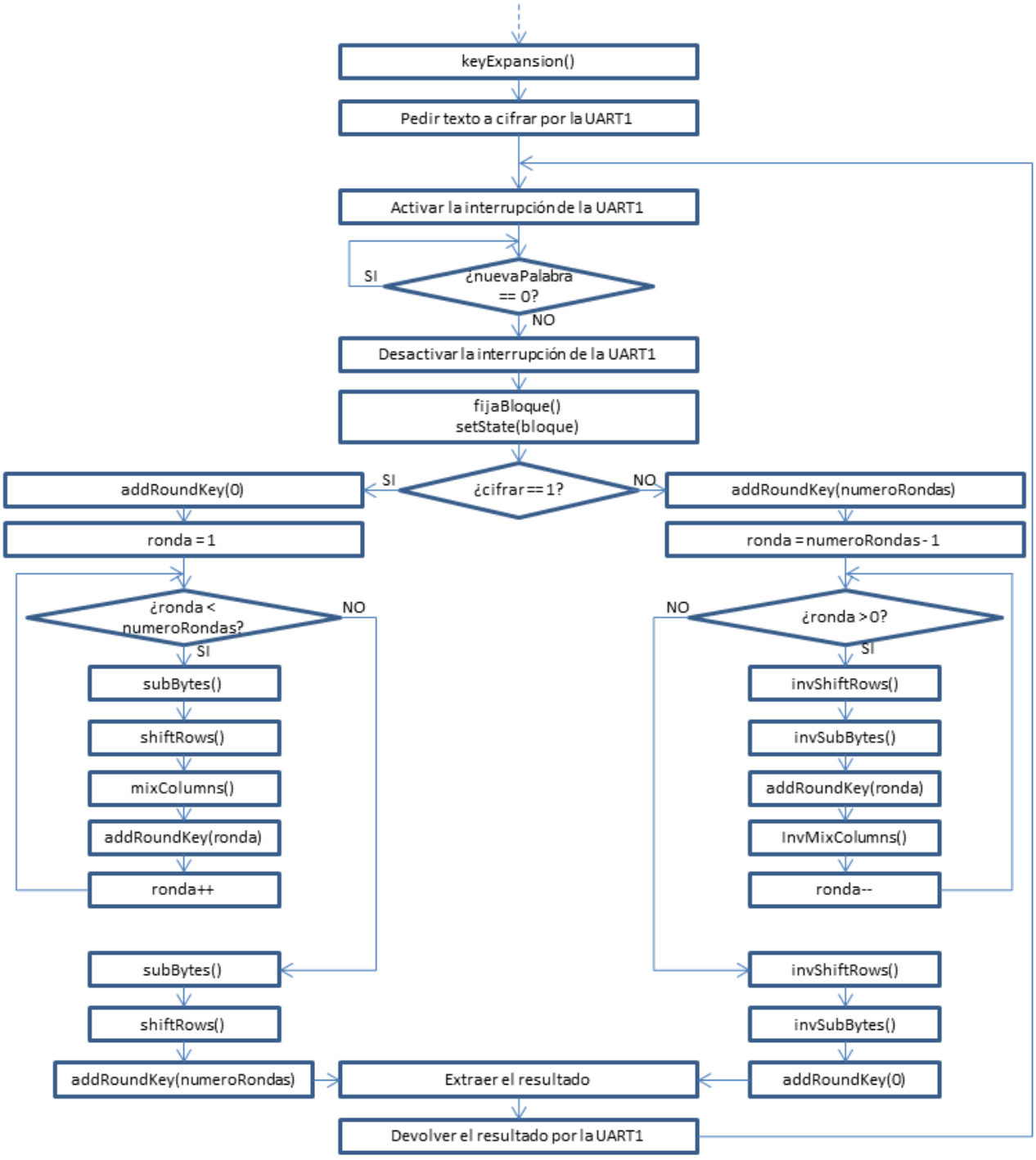

Figura 23. Fase de ejecución de AES.

Para la toma de datos se usan dos interrupciones distintas asociadas a la UART1 del microprocesador. Como se puede ver en la figura 24 su funcionamiento es similar. No obstante, el tamaño de clave variable y la distinta variable que recibe los datos obligan a que sean ligeramente diferentes.

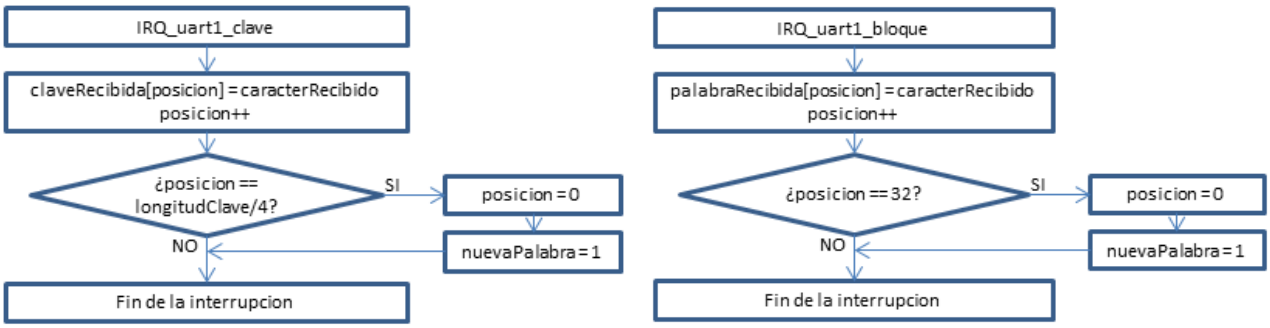

Figura 24. Interrupciones de recepción de datos.

Por tanto, tras averiguar si el usuario quiere cifrar o descifrar y tras obtener la clave, el programa permanecerá a la espera de bloques a cifrar o descifrar, los cuales devolverá cifrados o descifrados por el puerto serie.

4.1.2 Distribución del programa

El programa realizado se ha dividido en 4 diferentes archivos. Cada uno de estos archivos contiene funciones y variables clasificadas por su utilidad.

Los archivos del programa son AES.c, libreria_uart1.c y main.h. y en la tabla 6 se indica que funciones y variables acoge cada uno de ellos. Además ofrece una breve descripción sobre la utilidad y el funcionamiento de algunas de las funciones o variables.

aes.c	Variables	int sbox[256]	Tabla S-box.
		int isbox[256]	Tabla S-box inversa.
		estado [4][4]	Matriz de estado sobre la que se irán aplicando todas las operaciones durante el proceso de cifrado.
		roundKey[240]	Array con el conjunto de todas las subclaves de ronda.
		int Rcon[255]	Constantes Rcon.
	Funciones	setEstado(entrada[16])	Coloca los componentes del array de entrada en sus lugares dentro de la variable estado[4][4].
		getEstado()	Devuelve un puntero al array de la matriz de estado
		keyExpansion()	Crea todas las subclaves de ronda a partir del contenido de la variable clave del archivo main.c
		addRoundKey(ronda)	Realiza la operación AddRoundKey entre la matriz de estado y la correspondiente subclave de ronda depositando el resultado en la matriz de estado
		subBytes()	Estas operaciones se corresponden con las operaciones principales del algoritmo AES y sus inversas. Trabajan directamente sobre la matriz de estado y no necesitan ningún parámetro.
		invSubBytes()	
		shiftRows()	
		invShiftRows()	
		mixColumns()	
		invMixColumns()	

libreria_uart1.c	Variables	caracteresAEnviar[50]	Buffer de envío de datos por el puerto serie de 50 caracteres de capacidad.
	Funciones	iniUart1()	
		enviarCadena_uart1(cadena)	
		enviarSaltoDeCarro_uart1()	
main.c	Variables	cifrar	1 – cifrar, 0 – descifrar.
		longitudClave	
		clave[32]	Array que guarda la clave de cifrado. Su tamaño es el mayor posible, en caso de que la clave no sea de la mayor longitud (256 bits) las últimas posiciones del array se ignorarán.
		claveRecibida[64]	Almacena los caracteres recibidos como clave.
		palabraRecibida[32]	Almacena los caracteres recibidos como bloque a cifrar.
		bloque[16]	Array que guarda el bloque a cifrar tras averiguar el significado de los caracteres recibidos.
		palabraProcesada[16]	Guarda el bloque resultante del proceso de cifrado/descifrado.
		numeroRondas	
		nuevaPalabra	Indica cuando se ha recibido todo un bloque por el puerto serie.
	Funciones	fijaBloque()	Averigua el valor real de los caracteres ASCII recibidos como bloque de cifrado y lo almacena en la variable bloque.
		fijaClave()	Averigua el valor real de los caracteres ASCII recibidos como clave y lo almacena en la variable clave.
		leeCifra(caracter)	Devuelve el valor real de un carácter ASCII
		leeChar(valor)	Devuelve el carácter ASCII de un valor numérico
		main()	Método main del programa. Comienza a partir de él.

Tabla 6. Funciones y variables de la primera versión del programa.

En conjunto, estos cuatro archivos nos permiten ejecutar el algoritmo AES con un funcionamiento que sigue paso por paso las especificaciones de Daemen y Rijmen [18] explicadas en el capítulo 2.2.

4.1.3 Puesta en funcionamiento del programa

Hasta ahora hemos descrito grosso modo el funcionamiento del programa, pero no hemos explicado como se puede llegar a ejecutar.

Este programa se puede ejecutar con multitud de plataformas, no obstante en este proyecto se ha ejecutado en utilizando un ordenador con Windows XP (Que incluye el programa HyperTerminal para comunicarse por el puerto serie) y el cable de comunicación serie – USB TTL-232R-3V3. Por esto a continuación explicaremos como ejecutarlo con estos elementos.

1. Instalar el software Keil uVision 4 y su licencia de uso.
2. Instalar los drivers del JTAG, los cuales se pueden obtener en la página del fabricante [19].
3. Instalar los drivers del cable de comunicación serie – USB (TTL-232R-3v3) siguiendo la guía de instalación [16].
4. Una vez contemos con todo el software necesario, habrá que conectar el microprocesador a la alimentación mediante un cable adaptador USB y al JTAG mediante una faja de conexión en paralelo. Es importante conectar también el JTAG al ordenador mediante un adaptador USB para que se pueda comunicar con el mismo.

5. También habrá que conectar los cables del USB-puerto serie (El cable amarillo al pin 10 de la placa, UART1-Tx, y el cable naranja al pin 11, UART1-Rx)
6. Ejecutar AES mediante uVision.
   6.1 Crear una carpeta y copiar dentro los archivos del programa.
   6.2 Crear un proyecto en uVision y adjuntar esos archivos.
   6.3 Acceder al menú Target Options. En Debug marcar Use Ulink ARM Debugger y en Utilities seleccionar "use Target Driver for Flash Programming" "ULINK ARM Debugger" marcar Update Target before Debugging y pulsar en Settings. En programming algorithm debe estar LPC2000 IAP2 64kB Flash (si no está, habrá que añadirlo mediante Add) y se debe seleccionar Program, Verify y Reset and Run.
   6.4 Presionar Build y Rebuild para compilar el proyecto y después Download para descargarlo al micro y ejecutarlo. Ahora AES está ejecutándose en el micro.
7. Crear una conexión por el puerto serie mediante el programa HyperTerminal o algún otro, seleccionando el puerto serie utilizado y configurando la conexión a 9600 bits por segundo, con 8 bits de datos, paridad ninguna, 1 bit de parada y control de flujo ninguno.

Tras seguir estos pasos ya tenemos el algoritmo AES ejecutándose en el microprocesador y una manera de comunicarnos con él para facilitarle tanto los parámetros de cifrado como los caracteres a cifrar/descifrar y para recibir los caracteres cifrados/descifrados.

Los pasos 1, 2 y 3 solo sería necesario aplicarlos la primera vez, ya que para el resto de ocasiones el software ya estaría instalado.

4.2 *Mejoras de rendimiento*

Después de implementar el algoritmo AES y de ejecutarlo sobre el microprocesador, el principal objetivo es mejorar su rendimiento, ya bien sea aprovechando características del microprocesador o del mismo algoritmo.

Nuestro objetivo no es únicamente mejorar el programa, si no principalmente mejorar el propio algoritmo para que sea más eficiente y poder utilizar esas mejoras en futuras aplicaciones. De esta manera estas mejoras podrán ser implementadas en cualquier otra implementación del algoritmo. Es por esto que nos hemos centrado en mejorar las funciones propias del algoritmo y no en la eficiencia de la recepción o devolución de los datos u otros puntos del programa que estén dentro de la estructura del algoritmo AES.

Para analizar el algoritmo y para comprobar estas mejoras se han efectuado mediciones de tiempo. El programa Keil uVision4 aporta una herramienta que nos permite simular el funcionamiento del programa y nos ayuda a saber cuanto tiempo tarda en efectuar todas y cada una de las operaciones. No obstante, antes de utilizar esta herramienta se modificó el código del programa para tomar varias medidas reales utilizando uno de los temporizadores que aporta el LPC2132 y se compararon con el valor teórico que habíamos obtenido gracias a uVision. El resultado de estas medidas era prácticamente exacto al obtenido mediante el software, lo cual nos permitió continuar realizando medidas de una manera más eficaz al no tener la necesidad de modificar el código del programa para ello.

4.2.1 Análisis del algoritmo AES

Antes de ponernos a realizar mejoras sin tener un orden ni un punto sobre el que centrarnos se ha realizado un análisis del algoritmo. Este análisis se puede dividir en dos partes.

En la primera estudiamos el tiempo de ejecución de cada una de las funciones y lo comparamos con el global y con el del resto para ver si merece la pena incidir más en alguna función que en otra.

En la segunda se analiza como está programado para ver que tipo de mejoras darán una mayor eficiencia en tiempo de proceso que es lo que se busca.

4.2.1.1 Análisis de tiempos de AES

Como se acaba de decir, se han realizado mediciones de tiempos de cada una de las distintas funciones. Se ha medido el tiempo que el microprocesador está realizando operaciones referentes a cada una de las funciones si cifra un bloque y si cifra 10 bloques.

El motivo de medir también los tiempos cuando se cifran 10 bloques de cifrado es que por muchos bloques que se cifren, la operación *KeyExpansion* se realizará una sola vez. Por tanto conviene ver como varía el peso de la misma respecto al número de bloques de cifrado.

Los resultados de estas mediciones se pueden observar en la tabla 7.

	Un solo bloque de cifrado			**10 bloques de cifrado**		
Tamaño de clave	128 bits	192 bits	256 bits	128 bits	192 bits	256 bits
Extracción e inserción de datos	12.904µs	12.904µs	12.904µs	129.04µs	129.04µs	129.04µs
KeyExpansion	157.25µs	176.183µs	211.717µs	157.25µs	176.183µs	211.717µs
AddRoundKey	94.5µs	111.7 µs	128.9µs	945µs	1117µs	1289µs
SubBytes	73.333µs	88µs	102.667µs	733.33µs	880µs	1026.67µs
ShiftRows	13.333µs	16µs	18.667µs	133.33µs	160µs	186.67µs
MixColumns	89.550µs	109.45µs	129.35µs	895.50µs	1094.5µs	1293.5µs
InvShiftRows	13.333µs	16µs	18.667µs	133.33µs	160µs	186.67µs
InvSubBytes	73.333µs	88µs	102.667µs	733.33µs	880µs	10266.7µs
InvMixColumns	1241µs	1517µs	1793µs	12410µs	15170µs	17930µs

Tabla 7. Análisis de tiempos de AES.

Para obtener una idea más clara del peso de cada una de las operaciones se ha creado la figura 25, que representan de una manera más visual de estos valores.

En el eje Y de la misma los tiempos están representados en microsegundos y cada color simboliza el tiempo que el procesador se encuentra dentro de una función.

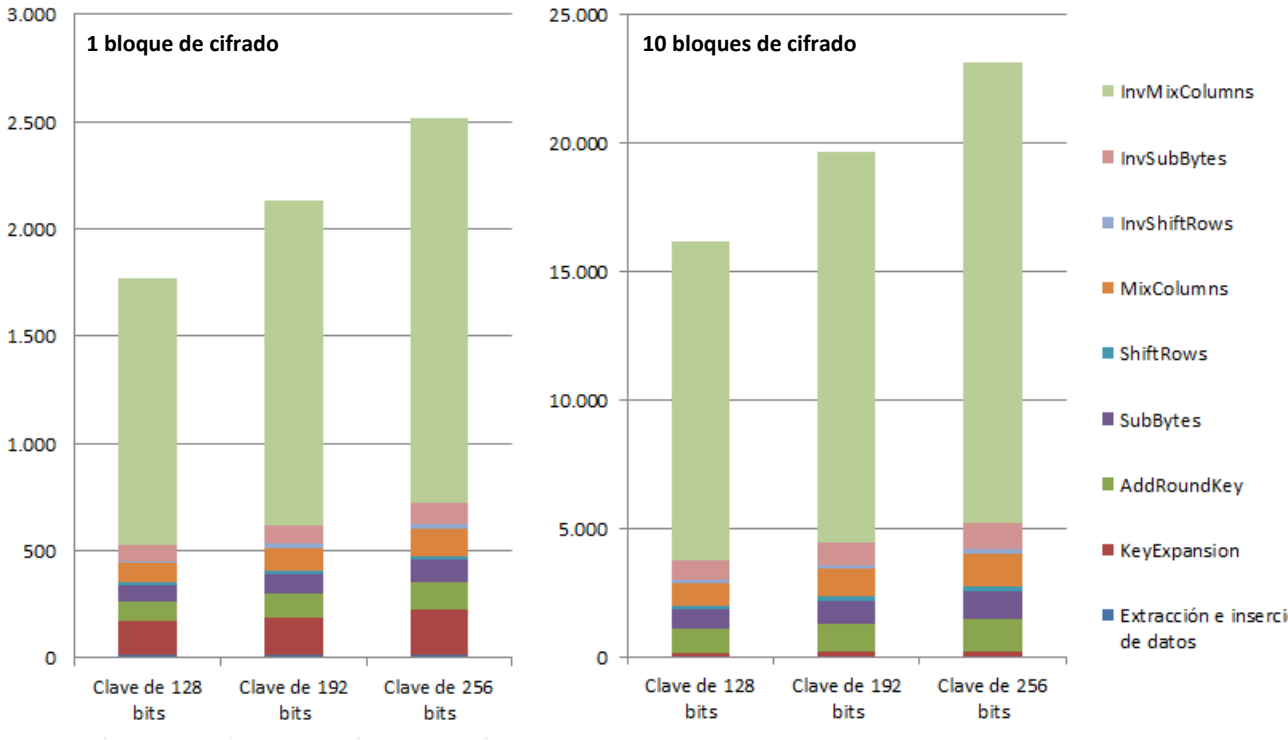

Figura 25. Análisis de tiempo de AES.

Se observa como el peso del tiempo de cada una de las funciones respecto del resto es independiente del tamaño de clave.

Además se puede ver claramente como la función que más peso tiene es InvMixColumns, ocupando tres cuartas partes del total. También se aprecia que la extracción e inserción de los datos y las funciones ShiftRows e InvShiftRows tienen muy poco peso.

Otro dato que podemos ver es que a pesar de que la función KeyExpansion tenga bastante peso cuando se cifra un solo bloque, cuando se cifran 10 o más su peso resulta insignificante.

Por esto se seguirán estos criterios:

1. La función InvMixColumns tendrá que ser fuertemente revisada para reducir al máximo el tiempo de proceso de la misma puesto que tiene demasiado peso computacional.
2. La extracción e inserción de datos y a las funciones ShiftRows e InvShiftRows serán revisadas pero sus mejoras son las menos prioritarias
3. En caso de que se ejecuten pocos bloques de cifrado la función KeyExpansion tiene bastante importancia. Por esto será revisada para reducir sus tiempos. No obstante, las reducciones de tiempos en esta función no tendrán mucha prioridad ya que en la mayoría de aplicaciones reales se cifrarán muchos bloques y el peso de KeyExpansion tenderá a cero.

4. El resto de funciones serán revisadas para reducir al máximo posible sus tiempos de proceso puesto que aunque no tienen un peso tan grande como InvMixColumns, siguen teniendo bastante peso.

4.2.1.2 Análisis del programa AES

Para lograr una mayor eficiencia computacional se ha analizado el código creado, las operaciones propias del algoritmo y las posibilidades de desarrollo que el microprocesador aportaba.

Se han descubierto varios puntos fuertes que se pueden explotar como la posibilidad de trabajar con palabras de 32 bits con la que cuenta nuestro microprocesador y puntos débiles que se pueden depurar como el uso de operaciones redundantes.

Por tanto se han descubierto diferentes tipos de mejoras que se pueden utilizar en distintas partes del programa y funciones del algoritmo.

Las mejoras que más se van a utilizar y que se espera que traigan mejores resultados son las siguientes:

- **Ejecución del algoritmo con 32 bits de palabra.** Esta es una de las mejoras que más aportaría a la reducción de tiempo. Se basa en utilizar toda la capacidad del microprocesador y ejecutar las operaciones cogiendo 32 bits en vez de 8, por lo que idealmente reduciría a la cuarta parte el número de operaciones realizadas.
- ***Loop unrolling*.** El *loop unrolling* o *loop unwinding* es una técnica de programación muy usada que se basa en desenrollar bucles para que no haya necesidad de realizar las operaciones que se realizan al índice contador del bucle o que dependen de él al estar precalculadas. Un inconveniente que tiene el *loop unrolling* es que al "desenrollar" los bucles se realiza mayor uso de memoria, por esto, solo se utilizará esta técnica para bucles cuyo "desenrollado" no conlleve un notable incremento de memoria
- **Eliminación y/o simplificación de operaciones repetidas.** Analizando minuciosamente las especificaciones de AES de Daemen y Rijmen se han encontrado algunas operaciones que se realizan varias veces. Eliminando estas operaciones se puede reducir el tiempo de una manera notable.
- **Eliminación de variables y operaciones asociadas.** En los códigos de programación suelen existir variables que son recursos del programador. Estas variables facilitan la programación pero hacen que haya más operaciones. Por tanto, en algunos casos, si se eliminan se gana en tiempo de cómputo global.
- **Eliminación de macros.** En el programa se han utilizado dos macros que resumen un conjunto de operaciones. Eliminando uno de ellos se conseguiría reducir el tiempo de proceso.

Estas son las técnicas que se utilizarán. A continuación se analizará como se implementan en las distintas partes del programa y que resultados dan.

4.2.2 Implementación de las mejoras

Tras analizar como y donde poder minimizar al máximo el tiempo de proceso del programa, en los siguientes apartados se procederá a implementar esas mejoras y analizar sus resultados.

Para ello empezaremos con una modificación estructural que nos permitirá utilizar toda la capacidad del microprocesador.

4.2.2.1 Utilización de toda la capacidad del microprocesador

Tras este apartado se irán comentando las múltiples mejoras que se han conseguido. No obstante el principal recurso para llegar hasta ellas ha sido el uso de toda la capacidad del microprocesador, por lo que en este apartado se explica como se ha logrado usar esta capacidad.

Analizando el algoritmo AES notamos que, como hemos explicado anteriormente, trabaja con operaciones a nivel de 8 bits, ya que cada uno de los elementos de la matriz de estado ocupa 8 bits. Sin embargo nosotros contamos con un microprocesador capaz de trabajar con palabras de 32 bits. Por tanto cada vez que el microprocesador realiza una operación con un único byte está desaprovechando gran parte de su capacidad. Nuestro objetivo principal será por tanto conseguir que el microprocesador trabaje con 32 bits en lugar de hacerlo con un único byte. Para hacer esto habrá que modificar ampliamente la estructura de nuestro programa.

En su primera versión, el cifrado se basaba en colocar en una matriz de cuatro por cuatro bytes los elementos de la matriz de estado y realizar una serie de operaciones sobre ella.

Lo primero que se puede llegar a pensar es que podríamos sustituir la matriz por un array de 4 variables de 32 bits, que deberían almacenar cada una de las filas de la matriz de estado. Sin embargo, nosotros optaremos por trabajar con una versión traspuesta de la matriz de estado, la cual quedará representada por un array de cuatro enteros que contendrán los valores de sus filas, ya que de esta manera se consiguen mejores resultados globalmente.

Por tanto si la distribución de los bytes de la matriz de estado anteriormente era esta:

$$estado[4][4] = \begin{bmatrix} 0 & 4 & 8 & C \\ 1 & 5 & 9 & D \\ 2 & 6 & A & E \\ 3 & 7 & B & F \end{bmatrix}$$

Actualmente será así:

$$estado[4] = \begin{bmatrix} 0 \\ 4 \\ 8 \\ C \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 9 \\ D \end{bmatrix} \begin{bmatrix} 2 \\ 6 \\ A \\ E \end{bmatrix} \begin{bmatrix} 3 \\ 7 \\ B \\ F \end{bmatrix}$$

Notar que los bytes de la matriz de estado se almacenarán en este orden.

$$estado[0] = 048C$$

$$estado[1] = 159D$$

$$estado[2] = 26AE$$

$$estado[3] = 37BF$$

A continuación pasaremos a describir todas las modificaciones que se han efectuado en el código y sus aportaciones en tiempo por orden de ejecución.

4.2.2.2 Mejoras en la extracción e inserción de los datos

En este apartado explicaremos la introducción de dos mejoras en la extracción de los datos y una en la inserción.

Modificar la matriz de estado naturalmente afecta al establecimiento de la misma (función *setEstado()* en nuestro código) y a la extracción de los datos (*getEstado()*). Es por ello que tenemos que cambiar estas funciones.

Una vez cambiada la función *getEstado()* y adaptado el código del método main, hemos aprovechado las características del mismo para desenrollar un bucle y así reducir los cálculos.

En programación los bucles son un recurso que se utiliza mucho, pero si nos atenemos a la eficiencia en tiempo de ejecución debemos hacerlo el menor número de veces posible, ya que introducen operaciones extra como ya hemos comentado con anterioridad.

En esta tabla se puede ver la eficiencia conseguida después de estos cambios. Aclaramos que la reducción de tiempo representa el tiempo que el microprocesador deja de necesitar para realizar las operaciones en la versión mejorada y el porcentaje que este representa respecto del tiempo que tardaba antes de las mejoras.

	Tiempo de inserción de datos.	**Tiempo de extracción de datos.**
Versión original.	4.834µs	8.070µs
Tras la mejora de 32 bits.	2.150µs	7.200µs
Tras *loop unrolling*.	N/A	5.333µs
Reducción de tiempo.	2.684µs (**55.53%**)	2.737µs (**33.92%**)

Tabla 8. Mejoras extracción e inserción de los datos.

Observamos que tras estas dos mejoras tanto la extracción como la inserción de datos mejoran notablemente aunque no era necesario conseguir demasiada reducción puesto que tienen muy poco peso comparadas con otras funciones.

4.2.2.3 Función KeyExpansion (keyExpansion())

El principal problema que encontramos a la hora de pasar la función KeyExpansion al formato de 32 bits es que al igual que la matriz de estado, las claves también tienen que estar traspuestas.

Ante él tenemos dos soluciones:

1. Realizar una adaptación a 32 bits del algoritmo de generación de claves y posteriormente trasponer los resultados.
2. Modificar el algoritmo de generación de claves para obtener directamente las claves traspuestas.

La opción uno es la más sencilla, ya que trasponer una matriz es un cálculo muy básico. Pero como lo que estamos buscando es la eficiencia tendremos que probar las dos opciones y ver cual es más efectiva.

Por tanto se ha tenido que diseñar el algoritmo de generación de claves traspuestas. Esta tarea no ha sido sencilla puesto que como se ve en la figura 26 la generación de claves utiliza palabras anteriores para la obtención de una nueva palabra y al estar traspuestas debe tomar parte de varias palabras distintas en cada paso. En la figura 26, las palabras sin trasponer aparecen en vertical y las palabras traspuestas serían el contenido de los rectángulos horizontales en rojo.

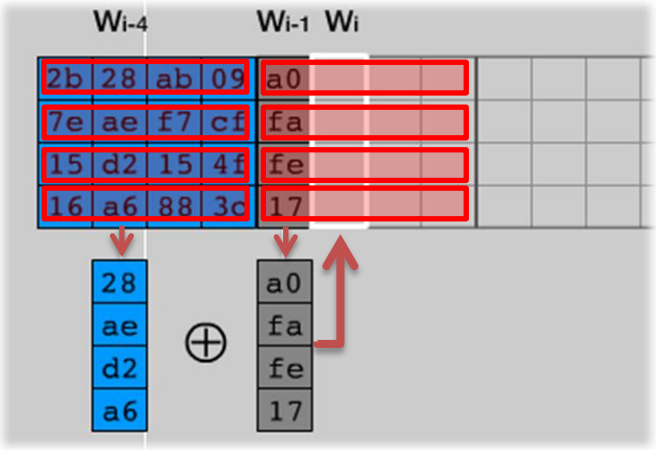

Figura 26. Generación de claves traspuestas.

Tras desarrollar estas dos soluciones obtenemos los resultados que aparecen en la tabla 9.

Tamaño de clave.	128 bits	192 bits	256 bits
Tiempo versión original.	157.250µs	176.183µs	211.717µs
Tiempo tras mejora Opción 1. (Tiempo trasposición)	67.017µs (29.951µs)	76.667µs (35.317µs)	96.350µs (40.684µs)
Tiempo tras mejora Opción 2.	41.067µs	63.383µs	57.517µs
Reducción de tiempo.	116.183µs (**73.88%**)	112.800µs (**64.02%**)	154.200µs (**72.83%**)

Tabla 9. Mejoras keyExpansion().

Los tiempos tras la mejora Opción 1 representan la suma del tiempo en generar las subclaves y el tiempo en trasponerlas. El tiempo que se tarda en trasponer las subclaves aparece debajo entre paréntesis.

Se puede ver como con la opción 1 se reduce enormemente el tiempo de proceso. No obstante, el tiempo que la función invierte en trasponer los resultados es muy alto.

Llegados a este punto es donde se puede apreciar la utilidad de realizar esa trasposición implícitamente tal y como se realiza en la opción 2. En este caso, el tiempo total es bastante similar al de la opción 1 sin contar la trasposición de los resultados.

Por tanto, como no invierte tiempo en trasponer los resultados, la opción 2 resulta mucho más eficiente consiguiendo llegar a invertir solamente la cuarta parte de tiempo que invertía antes de esta mejora.

También se puede notar como en el caso de 192 bits se obtienen peores resultados que en los casos de 128 y 256. Esto es debido a que se dificultan los cálculos porque cada 6 palabras hay que sumar la constante *rcon* pero las subclaves son de 4 palabras, y estos números no son múltiplo el uno del otro, de esta manera no es sencillo definir en que punto hay que sumar la constante.

4.2.2.4 Función AddRoundKey (addRoundKey(r))

Esta función se encargaba de mezclar la matriz de estado con la subclave de ronda, por lo que se apoya en la mejora aplicada a la función KeyExpansion en el apartado anterior.

Como en la mayoría de apartados, se han aplicado varias mejoras. Primero la adaptación al funcionamiento en 32 bits y a continuación *loop unrolling* y eliminación de operaciones sencillas.

La adaptación al funcionamiento se ha basado en sumar la subclave y la matriz de estado en palabras de 4 elementos en lugar de hacerlo de elemento en elemento.

El resultado de estas mejoras queda reflejado en la tabla 10.

Conviene notar que la función AddRoundKey es llamada un número de veces igual al número de rondas más uno, por lo que la reducción real será aproximadamente dicho número (11, 13 o 15) multiplicado por la reducción por ronda.

	Por ronda.	**Con clave de 128 bits.**	**Con clave de 192 bits.**	**Con clave de 256 bits.**
Tiempo versión original.	8.417µs	94.500µs	111.700µs	128.900µs
Tiempo tras mejora de 32 bits.	2.083µs	22.983µs	29.550µs	34.117µs
Tiempo tras *loop unrolling*.	1.183µs	15.083µs	17.851µs	20.617µs
Reducción de tiempo.	7.234µs (85.95%)	79.417µs (**84.04%**)	93.849µs (**84.02%**)	108.283µs (**84.01%**)

Tabla 10. Mejoras addRoundKey().

Podemos ver que la principal mejora de las aplicadas a la función Addroundkey ha sido la de ejecutar las operaciones a nivel de 32 bits. No obstante solo con las mejoras de *loop unrolling* y de operaciones sencillas se llega a aplicar una reducción de tiempo de casi el 50% en el caso de clave de 256 bits, lo cual es un porcentaje bastante alto.

Para concluir podemos decir que la versión original tardaba muy poco tiempo por lo que era más difícil conseguir mejoras notables. Aun así se ha llegado a una reducción de tiempo del 85,95%.

4.2.2.5 Función SubBytes (subBytes())

Antes de analizar las mejoras aplicadas vamos a recordar que la función SubBytes se encargaba de introducir confusión a todo el proceso de cifrado/descifrado realizando una sustitución de los bytes de la matriz de estado con los correspondientes dentro de una tabla S-Box.

Conociendo esto, la manera más lógica de aplicar una mejora de funcionamiento sería creando a partir de la tabla S-Box actual una de 32 bits para realizar 4 consultas por ronda. No obstante, esta tabla tendría que tener $2^{32} = 4.294.967.296$ diferentes entradas y ocuparía $32 \cdot 2^{32}$ bits, o sea, 16 GigaBytes.

El LPC2132 con el que estamos realizando este proyecto no dispone de tanta memoria pero aunque podría utilizarse una memoria externa no merecería la pena aumentar tanto el consumo de memoria por una reducción en tiempo de 7µs por ronda.

Por tanto, para adaptar la función a los 32 bits se ha modificado la tabla S-Box sin cambiar el número de entradas, aunque se han convertido todas sus entradas a valores enteros (de 32 bits) añadiendo ceros por la izquierda en lugar de los bytes almacenados hasta ahora. Con esto se pueden utilizar más fácil y rápidamente los valores de la tabla.

De esta manera, aunque la función mantenga el número de consultas de tabla en 16, se reduce el número de modificaciones de la matriz de estado de 16 a 4. Esto hace que aunque no de una manera tan eficiente como en otras funciones, la adaptación a los 32 bits vea el tiempo de ejecución considerablemente reducido.

Además aquí también se ha aplicado *loop unrolling* y la mejora de reducción de algunas operaciones sencillas.

Los resultados obtenidos tras estas mejoras quedan reflejados en la tabla 11.

	Por ronda.	Con clave de 128 bits.	Con clave de 192 bits.	Con clave de 256 bits.
Tiempo versión original.	7.333µs	73.333µs	88.000µs	102.667µs
Tiempo tras mejora de 32 bits.	4.333µs	43.333µs	52.000µs	60.667µs
Tiempo tras *loop unrolling*.	3.483µs	34.833µs	41.800µs	48.767µs
Reducción de tiempo	3.850µs (52.5%)	38.500µs (**52.5%**)	46.200µs (**52.5%**)	53.900µs (**52.5%**)

Tabla 11. Mejoras subBytes().

Como podemos ver se ha llegado a reducir el tiempo de proceso en un 52.5% sin necesitar el uso de gigabytes de memoria.

4.2.2.6 Función ShiftRows (shiftRows())

La función ShiftRows anteriormente se dedicaba a rotar las filas de la matriz de estado un número distinto de bytes en función del número de fila para añadir difusión. Ahora, como hemos traspuesto la matriz de estado, tendrá que rotar las columnas en lugar de las filas.

Rotar las columnas nos viene muy bien ya que cada una de ellas está almacenada en una variable de 32 bits y nuestro microprocesador tiene una operación que se encarga de rotar los bits de una variable.

Por tanto, la mejora obtenida por la adaptación al funcionamiento en 32 bits consistirá en sustituir los 16 movimientos anteriores por 4 rotaciones de enteros.

También se ha aplicado una mejora de funcionamiento basada en la eliminación de una variable y algunas operaciones simples que utilizaban esta variable.

Tras estas mejoras se han obtenido los resultados reflejados en la tabla 12.

	Por ronda.	Con clave de 128 bits.	Con clave de 192 bits.	Con clave de 256 bits.
Tiempo versión original.	1.333µs	13.333µs	16.000µs	18.667µs
Tiempo tras mejora de 32 bits.	1.117µs	11.667µs	14.000µs	16.333µs
Tiempo tras mejora de variable y *loop unrolling*.	0.883µs	8.833µs	10.600µs	12.367µs
Reducción de tiempo.	0.450 µs (33.76%)	4.500µs (**33.75%**)	5.400µs (**33.75%**)	6.300µs (**33.75%**)

Tabla 12. Mejoras shiftRows().

Se puede apreciar como en este apartado conseguimos una mejora de un 33.75%.

Este valor es menor que el conseguido en otros apartados, pero hay que tener en cuenta que en este caso el tiempo de la operación era un orden de magnitud inferior al de los anteriores por lo que conseguir mejorar el tiempo de proceso era bastante más difícil y mucho menos importante.

Una manera de ver esta dificultad de mejora es observar que el tiempo de llamada a la función es un 5% del tiempo de ejecución de la misma.

Así mismo, hay que tener en cuenta que nuestro microprocesador tarda más en efectuar una operación de rotación de bits que una de asignación de variable, y una de las mejoras de esta función se basa en sustituir 16 operaciones de sustitución por cuatro de rotación.

Por tanto, basándonos en la dificultad de mejora y en el escaso peso de esta función respecto al tiempo total de cómputo podemos decir que el 33.75% de reducción de tiempo roza si no llega a alcanzar el resultado óptimo.

4.2.2.7 Función MixColumns (mixColumns())

La función MixColumns ha sido fuertemente revisada y rediseñada para adaptarla al funcionamiento en 32 bits.

De lo explicado en el apartado 2.2.4.1.3. y las modificaciones realizadas debidas a la adaptación a 32 bits podemos concluir diciendo que el resultado de esta función es el mismo que el de aplicar las siguientes operaciones:

$$y_0 = 02 * x_0 \oplus 03 * x_1 \oplus x_2 \oplus x_3$$
$$y_1 = x_0 \oplus 02 * x_1 \oplus 03 * x_2 \oplus x_3$$
$$y_2 = x_0 \oplus x_1 \oplus 02 * x_2 \oplus 03 * x_3$$
$$y_3 = 03 * x_0 \oplus x_1 \oplus x_2 \oplus 02 * x_3$$

Siendo x_i las columnas de la matriz de estado traspuesta antes de la transformación MixColumns e y_i después de la misma y correspondiéndose el operador * al producto en $GF(2^8)$ realizado en paralelo byte a byte.

La manera más sencilla de calcular estas operaciones es utilizar las variables y_i como acumuladores y x_i para almacenar el producto $02 * x_i$. Esto se puede hacer en tres pasos que se resumen en la tabla 13.

Paso 1.	Paso 2.	Paso 3.
$\boldsymbol{y_0 = x_1 \oplus x_2 \oplus x_3}$	$x_0 = 02 * x_0$	$y_0 \oplus= x_0 \oplus x_1$
$\boldsymbol{y_1 = x_0 \oplus x_2 \oplus x_3}$	$x_1 = 02 * x_1$	$y_1 \oplus= x_1 \oplus x_2$
$\boldsymbol{y_2 = x_0 \oplus x_1 \oplus x_3}$	$x_2 = 02 * x_2$	$y_2 \oplus= x_2 \oplus x_3$
$\boldsymbol{y_3 = x_0 \oplus x_1 \oplus x_2}$	$x_3 = 02 * x_3$	$y_3 \oplus= x_0 \oplus x_3$

Tabla 13. Pasos de mixColumns().

La adaptación de la función MixColumns al funcionamiento en 32 bits mediante estos tres pasos es la principal mejora aplicada en este apartado. No obstante, no estaría completa si no se adaptara también el macro *xtime(x)*.

El macro *xtime(x)* se encarga de realizar el producto de x por 2 en $GF(2^8)$. Hasta ahora funcionaba con palabras de 8 bits. No obstante en esta mejora se ha adaptado para que funcione también con palabras de 32 bits y así podemos realizar la operación una sola vez en lugar de 4 y no es necesario separar y volver juntar los bytes de cada columna.

Por último se ha añadido una tercera mejora a esta función consistente en la eliminación de 4 bucles, una variable y operaciones asociadas a los mismos gracias al *loop unrolling*.

Los resultados de estas tres mejoras se pueden ver en la tabla 14.

	Por ronda.	Con clave de 128 bits.	Con clave de 192 bits.	Con clave de 256 bits.
Tiempo versión original.	9.950µs	89.550µs	109.450µs	129.350µs
Tiempo tras mejora de 32 bits.	11.267µs	101.400µs	123.933µs	146.467µs
Tiempo tras mejora de xtime(x).	7.233µs	66.000µs	80.667µs	95.333µs
Tiempo tras *loop unrolling*.	4.433µs	39.900µs	48.767µs	57.633µs
Reducción de tiempo.	5.517µs (55.45%)	49.650µs (**55.44%**)	60.683µs (**55.44%**)	71.717µs (**55.44%**)

Tabla 14.Mejoras mixColumns().

Podemos ver que la unión de estas tres mejoras conducen a la reducción de un **55.44%** del tiempo de mixColumns, lo que puede llegar a ser 71.717µs. Para apreciar la magnitud del tiempo reducido resaltaremos que este tiempo que ahora nos ahorramos es casi el mismo tiempo que se tarda en ejecutar las operaciones AddRoundKey, SubBytes y ShiftRows juntas

Además se aprecia que la mejora de *xtime(x)* ha sido clave para permitir que la mejora de 32 bits aporte sus beneficios y que *loop unrolling* ha hecho posible aprovechar todo el potencial de la ejecución en 32 bits.

4.2.2.8 Función InvShiftRows (invShiftRows())

Como la función ShiftRows rotaba las columnas de la matriz de estado traspuesta, su función inversa consistirá en rotar dichas columnas en sentido opuesto.

Por tanto las mejoras aplicadas a la función InvShiftRows son bastante similares a las de la función ShiftRows basándonos en la adaptación al funcionamiento en 32 bits, en el *loop unrolling* y en la eliminación de una variable y de operaciones sencillas.

Los resultados de estas mejoras aparecen en la tabla 15.

	Por ronda.	Con clave de 128 bits.	Con clave de 192 bits.	Con clave de 256 bits.
Tiempo versión original.	1.333µs	13.333µs	16.000µs	18.667µs
Tiempo tras mejora de 32 bits.	1.117µs	11.667µs	14.000µs	16.333µs
Tiempo tras mejora de variable y *loop unrolling*.	0.883µs	8.833µs	10.600µs	12.367µs
Reducción de tiempo.	0.450 µs (33.76%)	4.500µs (**33.75%**)	5.400µs (**33.75%**)	6.300µs (**33.75%**)

Tabla 15. Mejoras invShiftRows().

Como podemos ver, las mejoras aplicadas nos permiten obtener los mismos resultados que en la función ShiftRows, llegando a conseguir un 33.75% de reducción de tiempo gracias a las mejoras aplicadas.

Esta reducción podría parecer pequeña pero hay que recordar que esta función tiene muy poco peso en el tiempo de proceso de todo el descifrado.

4.2.2.9 Función InvSubBytes (invSubBytes())

En el apartado 2.2.4.2.1. vimos que la única diferencia práctica entre la función SubBytes y la función InvSubBytes era que los bytes se sustituían en una tabla inversa S-Box en lugar de una tabla normal S-box.

Por tanto en la función InvSubBytes podremos aplicar mejoras muy similares a las aplicadas en la función SubBytes teniendo que convertir para ello la tabla iS-Box en una tabla de valores enteros (32 bits) añadiendo ceros por la izquierda a sus valores.

Tras aplicar nuestras mejoras se registraron los resultados obtenidos en la tabla 16.

	Por ronda.	Con clave de 128 bits.	Con clave de 192 bits.	Con clave de 256 bits.
Tiempo versión original.	7.333µs	73.333µs	88.000µs	102.667µs
Tiempo tras mejora de 32 bits.	4.333µs	43.333µs	52.000µs	60.667µs
Tiempo tras *loop unrolling*.	3.483µs	34.833µs	41.800µs	48.767µs
Reducción de tiempo	3.850µs (52.5%)	38.500µs (**52.5%**)	46.200µs (**52.5%**)	53.900µs (**52.5%**)

Tabla 16. Mejoras invSubBytes().

Los resultados nos muestran que las mejoras aplicadas reducen el tiempo de proceso en el mismo porcentaje (52.5%) sin aumentar en gigabytes la memoria utilizada.

Conviene comentar que el tiempo de ejecución podría verse más reducido si se utilizara una tabla de 32 bits de entrada, aunque esta tendría un tamaño de 16 gigabytes, demasiado grande para esta aplicación.

4.2.2.10 Función InvMixColumns (invMixColumns())

A continuación se describirán las mejoras aplicadas a la función InvMixColumns, así como los resultados que han aportado y una explicación sobre por qué se aplican de esta manera.

Las primeras tres mejoras aplicadas son muy similares a las aplicadas en la función MixColumns y se basan en la ejecución en palabras de 32 bits, la mejora del macro *xtime(x)* y en técnicas de *loop unrolling*.

Sin embargo, tras estas 3 mejoras se obtenía una considerable reducción en porcentaje pero la función seguía siendo demasiado pesada. Esto es debido a que la función InvMixColumns es la función que conlleva cálculos más complicados, por lo que ha sido la función que más se ha revisado y la que posiblemente más se haya llegado a optimizar.

Recordamos que para realizar la función MixColumns bastaba con aplicar a la matriz de estado estas operaciones:

$$y_0 = 02 * x_0 \oplus 03 * x_1 \oplus 01 * x_2 \oplus 01 * x_3$$
$$y_1 = 01 * x_0 \oplus 02 * x_1 \oplus 03 * x_2 \oplus 01 * x_3$$
$$y_2 = 01 * x_0 \oplus 01 * x_1 \oplus 02 * x_2 \oplus 03 * x_3$$
$$y_3 = 03 * x_0 \oplus 01 * x_1 \oplus 01 * x_2 \oplus 02 * x_3$$

Siendo x_i las columnas de la matriz de estado traspuesta antes de la transformación MixColumns e y_i después de la misma y correspondiéndose el operador * al producto en $GF(2^8)$ realizado en paralelo byte a byte.

El producto en el campo de Galois es una operación que con coeficientes altos adquiere una relativa complicación. Afortunadamente, en ese caso los coeficientes de este producto (Marcados en rojo oscuro) eran solamente 1, 2 o 3, lo cual eludía complicaciones.

Si volvemos a la función InvMixColumns, podemos observar como en este caso las operaciones a aplicar son estas:

$$y_0 = 14 * x_0 \oplus 11 * x_1 \oplus 13 * x_2 \oplus 09 * x_3$$
$$y_1 = 09 * x_0 \oplus 14 * x_1 \oplus 11 * x_2 \oplus 13 * x_3$$
$$y_2 = 13 * x_0 \oplus 09 * x_1 \oplus 14 * x_2 \oplus 11 * x_3$$
$$y_3 = 11 * x_0 \oplus 13 * x_1 \oplus 09 * x_2 \oplus 14 * x_3$$

Mientras que para el cifrado en la mitad de los casos los coeficientes no nos exigían realizar operaciones en $GF(2^8)$ y cuando eran necesarias eran con coeficientes bajos (2 o 3), para el descifrado son necesarias en todos los casos y con coeficientes más altos (9, 11, 13 o 14).

Antes de la mejora las operaciones en $GF(2^8)$ con coeficientes elevados se realizaban con un nuevo macro llamado *multiply(x, y)* que añadía mucho tiempo de proceso.

En un primer momento se pensó en mejorar ese macro haciendo que funcionara a nivel de 32 bits en lugar de 8 al igual que se hizo con el macro *xtime(x)*. Esta modificación daba buenos resultados pero con ella InvMixColumns seguía siendo demasiado pesada. Por tanto se optó por tratar de eliminar el macro *multiply(x, y)* y de reducir el número de operaciones que utilizaran *xtime(x)*.

Para eliminar el uso de este macro y de operaciones redundantes se ha tratado de simplificar las operaciones a aplicar separando los coeficientes en potencias de dos para poder hacer uso de *xtime(x)* en lugar de *multiply(x, y)*. De esta manera hemos obtenido estas ecuaciones:

$$y_0 = 8 * (x_0 + x_1 + x_2 + x_3) + 4 * (x_0 + x_2) + 2 * (x_0 + x_1) + (x_1 + x_2 + x_3)$$
$$y_1 = 8 * (x_0 + x_1 + x_2 + x_3) + 4 * (x_1 + x_3) + 2 * (x_1 + x_2) + (x_0 + x_2 + x_3)$$
$$y_2 = 8 * (x_0 + x_1 + x_2 + x_3) + 4 * (x_0 + x_2) + 2 * (x_2 + x_3) + (x_0 + x_1 + x_3)$$
$$y_3 = 8 * (x_0 + x_1 + x_2 + x_3) + 4 * (x_1 + x_3) + 2 * (x_3 + x_0) + (x_0 + x_1 + x_2)$$

Gracias a ello se ha descubierto que los coeficientes de InvMixColumns se pueden descomponer en una suma de los coeficientes de MixColumns y de unos coeficientes más sencillos:

$$[InvMixColumns] = [A] + [MixColumns]$$

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} = \begin{bmatrix} 12 & 8 & 12 & 8 \\ 8 & 12 & 8 & 12 \\ 12 & 8 & 12 & 8 \\ 8 & 12 & 8 & 12 \end{bmatrix} + \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Por tanto podemos aprovechar las operaciones usadas para calcular la matriz $[MixColumns]$ y beneficiarnos de que la matriz $[A]$ no es más que:

$$[A] = \begin{bmatrix} 2^3 & 2^3 & 2^3 & 2^3 \\ 2^3 & 2^3 & 2^3 & 2^3 \\ 2^3 & 2^3 & 2^3 & 2^3 \\ 2^3 & 2^3 & 2^3 & 2^3 \end{bmatrix} + \begin{bmatrix} 2^2 & 0 & 2^2 & 0 \\ 0 & 2^2 & 0 & 2^2 \\ 2^2 & 0 & 2^2 & 0 \\ 0 & 2^2 & 0 & 2^2 \end{bmatrix}$$

Después de este proceso hemos averiguado que podemos realizar estas operaciones en 5 sencillos pasos que se resumen en las siguientes tablas.

Los primeros 3 pasos, iguales a los de MixColumns, nos dan la matriz $[MixColumns]$.

Paso 1.	Paso 2.	Paso 3.
$y_0 = x_1 \oplus x_2 \oplus x_3$	$x_0 = 02 * x_0$	$y_0 \oplus= x_0 \oplus x_1$
$y_1 = x_0 \oplus x_2 \oplus x_3$	$x_1 = 02 * x_1$	$y_1 \oplus= x_1 \oplus x_2$
$y_2 = x_0 \oplus x_1 \oplus x_3$	$x_2 = 02 * x_2$	$y_2 \oplus= x_2 \oplus x_3$
$y_3 = x_0 \oplus x_1 \oplus x_2$	$x_3 = 02 * x_3$	$y_3 \oplus= x_0 \oplus x_3$

Tabla 17. Pasos InvMixColumns parte 1.

Los siguientes dos pasos calcularán $4 * (x_0 + x_2)$, $4 * (x_1 + x_3)$ y $8 * (x_0 + x_1 + x_2 + x_3)$ en el cuarto paso y lo sumarán pertinentemente en el quinto.

Paso 4.	Paso 5.
$x_0 = 02 * (x_0 + x_2)$	$y_0 \oplus= x_0 \oplus x_2$
$x_1 = 02 * (x_1 + x_3)$	$y_1 \oplus= x_1 \oplus x_2$
	$y_2 \oplus= x_0 \oplus x_2$
$x_2 = 02 * (x_0 + x_1)$	$y_3 \oplus= x_1 \oplus x_2$

Tabla 18. Pasos InvMixColumns parte 2.

Por tanto las mejoras aplicadas a la InvMixColumns han sido la adaptación al funcionamiento en 32 bits mediante el uso de palabras de 32 bits, la mejora del macro xtime(x), la eliminación de multiply(x, y) y de cálculos redundantes y la mejora de eliminación de bucles y variables prescindibles.

Tras su aplicación se han obtenido los resultados mostrados en la tabla 19.

	Por ronda.	Con clave de 128 bits.	Con clave de 192 bits.	Con clave de 256 bits.
Tiempo versión original.	138.117µs	1.241ms	1.517ms	1.793ms
Tiempo tras mejora de 32 bits.	199.450µs	1.793ms	2.192ms	2.590ms
Tiempo tras mejora de xtime(x) y multiply(x, y).	54.183µs	487.083µs	595.316µs	703.550µs
Tiempo tras eliminación de multiply(x, y) y de cálculos redundantes.	11.083µs	99.183µs	121.216µs	143.250µs
Tiempo tras mejora de *loop unrolling*.	7.650µs	68.283µs	83.450µs	98.616µs
Reducción de tiempo.	130.467µs (94.46%)	1.173ms (**94.50%**)	1.434ms (**94.50%**)	1.694ms (**94.50%**)

Tabla 19. Mejoras invMixColumns().

Los resultados obtenidos nos dan una reducción del **94.50%** del tiempo de proceso. Es una mejora muy grande, sobre todo si nos damos cuenta de que ese porcentaje era sobre un tiempo muy grande, lo cual se transforma en una mejora de milisegundos (Cuando el resto de operaciones no llegaban a tardar más de 200 microsegundos).

Además podemos apreciar como la suma de la adaptación a los 32 bits y la mejora de xtime() resulta de gran importancia, aunque la que mayor contribuye a la reducción es la mejora de la eliminación de multiply(x, y) y de cálculos redundantes, llegando a reducir a la quinta parte el tiempo de proceso.

Estas mejoras han sido claves para una ejecución eficiente del proceso de descifrado ya que antes de ellas era demasiado pesado por culpa de la función InvMixColumns.

4.2.3 Rendimiento tras la aplicación de todas las mejoras

Hasta ahora hemos descrito las distintas mejoras que hemos aplicado y sus consecuencias en el rendimiento de cada una de las funciones. En este apartado vamos a analizar sus consecuencias en el tiempo total de cifrado o descifrado de un bloque desde la inserción a la extracción de los datos.

Además de los tiempos anteriormente analizados aquí también vamos a incluir los motivados por la gestión de las distintas funciones, aunque realmente apenas afectan al resultado.

De esta forma obtenemos los siguientes resultados para el análisis de la actuación con un bloque de cifrado:

	Cifrado de un bloque.			**Descifrado de un bloque.**		
	128 bits de clave.	192 bits de clave.	256 bits de clave.	128 bits de clave.	192 bits de clave.	256 bits de clave.
Tiempo original.	460.366µs	534.133µs	624.500µs	1.612ms	1.999ms	2.288ms
Tiempo tras mejoras.	165.249µs	207.934µs	222.434µs	192.249µs	240.934µs	261.434µs
Reducción de tiempo.	295.117µs (64.10%)	326.199µs (61.07%)	402.066µs (64.38%)	1.419ms (88.02%)	1.759ms (87,99%)	2.027ms (88.59%)

Tabla 20. Tiempo total cifrando un bloque.

O con 10 bloques de cifrado:

	Cifrado de 10 bloques.			**Descifrado de 10 bloques.**		
	128 bits de clave.	192 bits de clave.	256 bits de clave.	128 bits de clave.	192 bits de clave.	256 bits de clave.
Tiempo original.	3.188ms	3.756ms	4.339ms	14.702ms	18.409ms	20.979ms
Tiempo tras mejoras.	1.283ms	1.509ms	1.707ms	1.553ms	1.839ms	2.097ms
Reducción de tiempo.	1.905ms (**59.76%**)	2.247ms (**59.82%**)	2.632ms (**60.66%**)	13.149ms (**89.43%**)	16.570ms (**90.01%**)	18.882ms (**90.00%**)

Tabla 21. Tiempo total cifrando 10 bloques.

Podemos notar como cifrando (o descifrando) un solo bloque, la reducción del tiempo de proceso en porcentaje varía en función del tamaño de clave. Esto es debido a que la función keyExpansion se ha reducido en mayor o menor porcentaje dependiendo del tamaño de clave y cuando solo se cifra/descifra un bloque esta función adquiere mayor peso.

Para visualizar la aportación de cada una de las distintas funciones vamos a ver en un primer momento el resultado de las mejoras durante el cifrado de un solo bloque de 16 bytes de una manera gráfica. Conviene notar que en las siguientes gráficas el eje Y representa el tiempo de ejecución en microsegundos.

En la figura 27 cada uno de los colores representa el tiempo que tarda cada una de las funciones del algoritmo en ser realizada por el microprocesador. Estos tiempos se muestran apilados para además dar una idea del tiempo global. Gracias a esto se puede ver el peso de cada una de las funciones dentro del total y lo que ha supuesto su mejora.

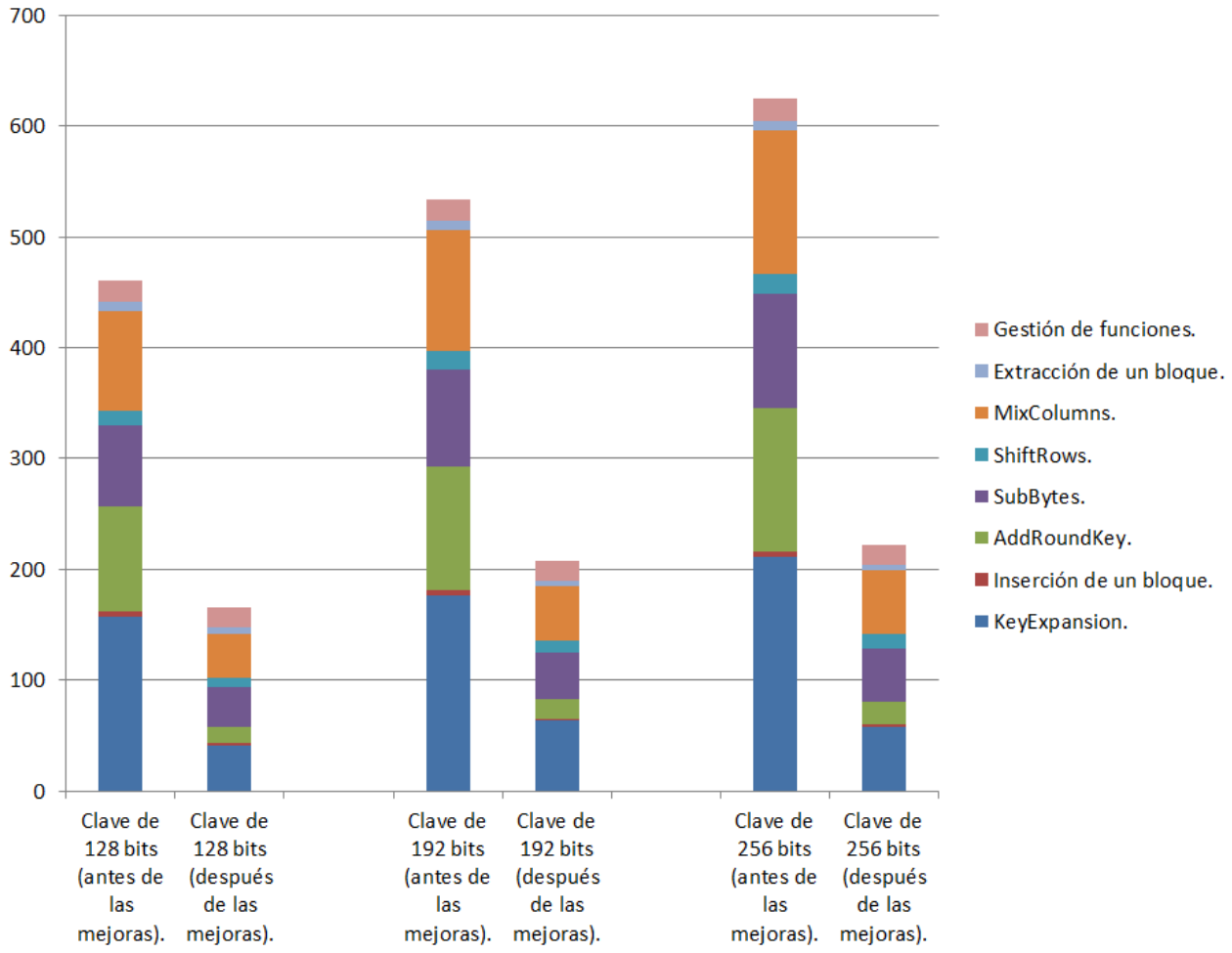

Figura 27. Rendimiento total de las mejoras cifrando un bloque.

En la gráfica se ha colocado una columna para cada tamaño de clave, representando el tiempo de proceso antes de las mejoras al lado del tiempo de proceso después de las mismas.

Aproximadamente se ha reducido el tiempo de cifrado de un bloque en un 65%, llegándolo a dejar en una cifra que se sitúa en torno a los 222 µs en el peor caso.

Se puede ver además que las funciones que aparentemente tienen más peso en el cifrado de datos son KeyExpansion, AddRoundKey, SubBytes y MixColumns, justo las funciones en las que gracias al análisis previo se han centrado las mejoras para llegar a obtener una mayor mejora en rendimiento.

No obstante, por muchos bloques que se cifraran, si la clave fuera la misma en todos ellos (como suele ser normal) la función KeyExpansion se ejecutaría una sola vez, por lo que estos resultados no serían del todo correctos. Por esto se han realizado mediciones de tiempos en un cifrado de 10 bloques distintos con la misma clave de codificación.

Los resultados se muestran en la figura 28.

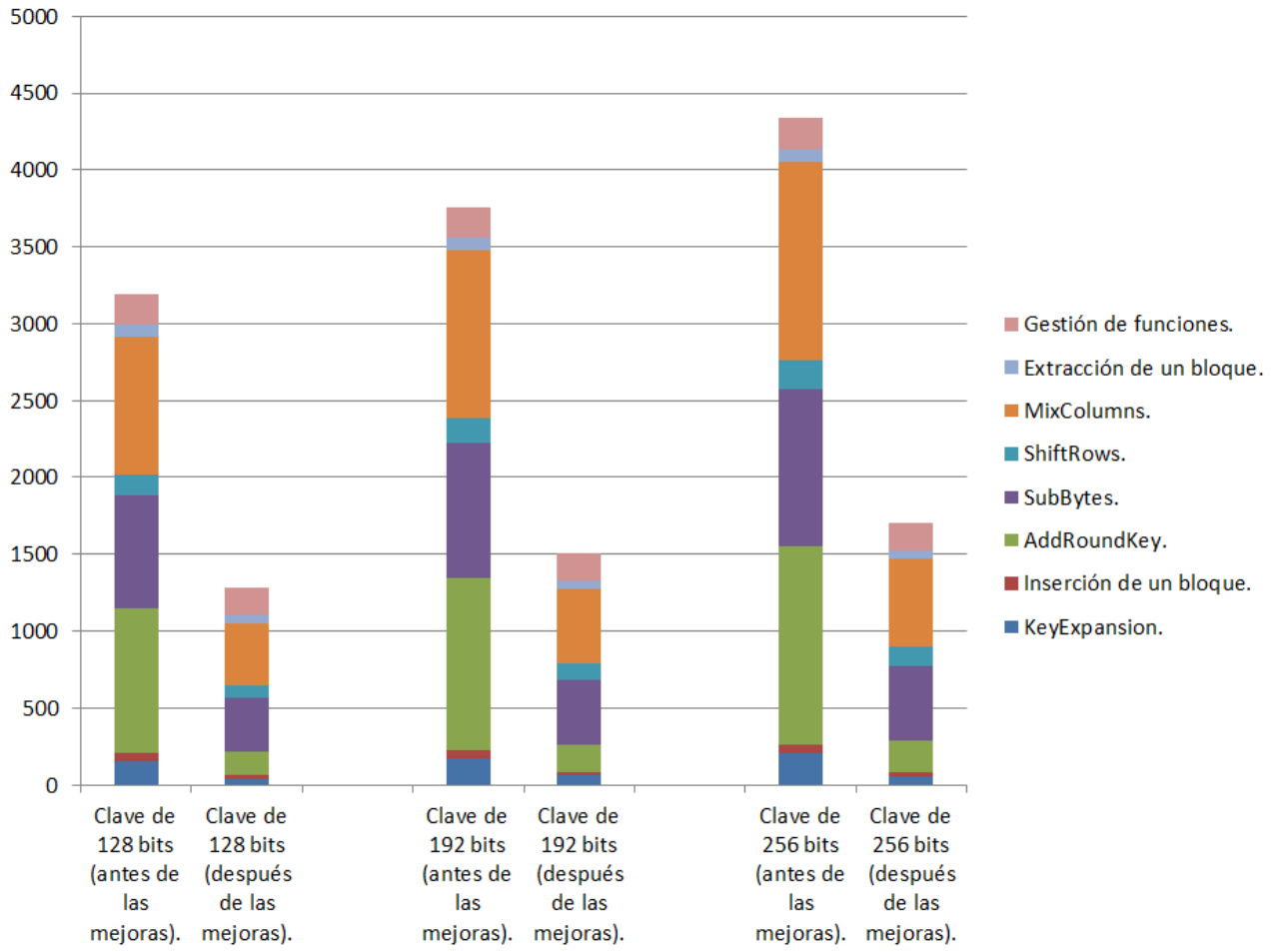

Figura 28. Rendimiento total de las mejoras cifrando 10 bloques.

Aquí se puede observar como la función KeyExpansion pierde mucho peso dejándoselo a las funciones AddRoundKey, SubBytes y MixColumns, las cuales habían sido reducidas en unos porcentajes de un 84%, 52% y 55% respectivamente

Para analizar la eficiencia durante el proceso de descifrado observaremos la figura 29.

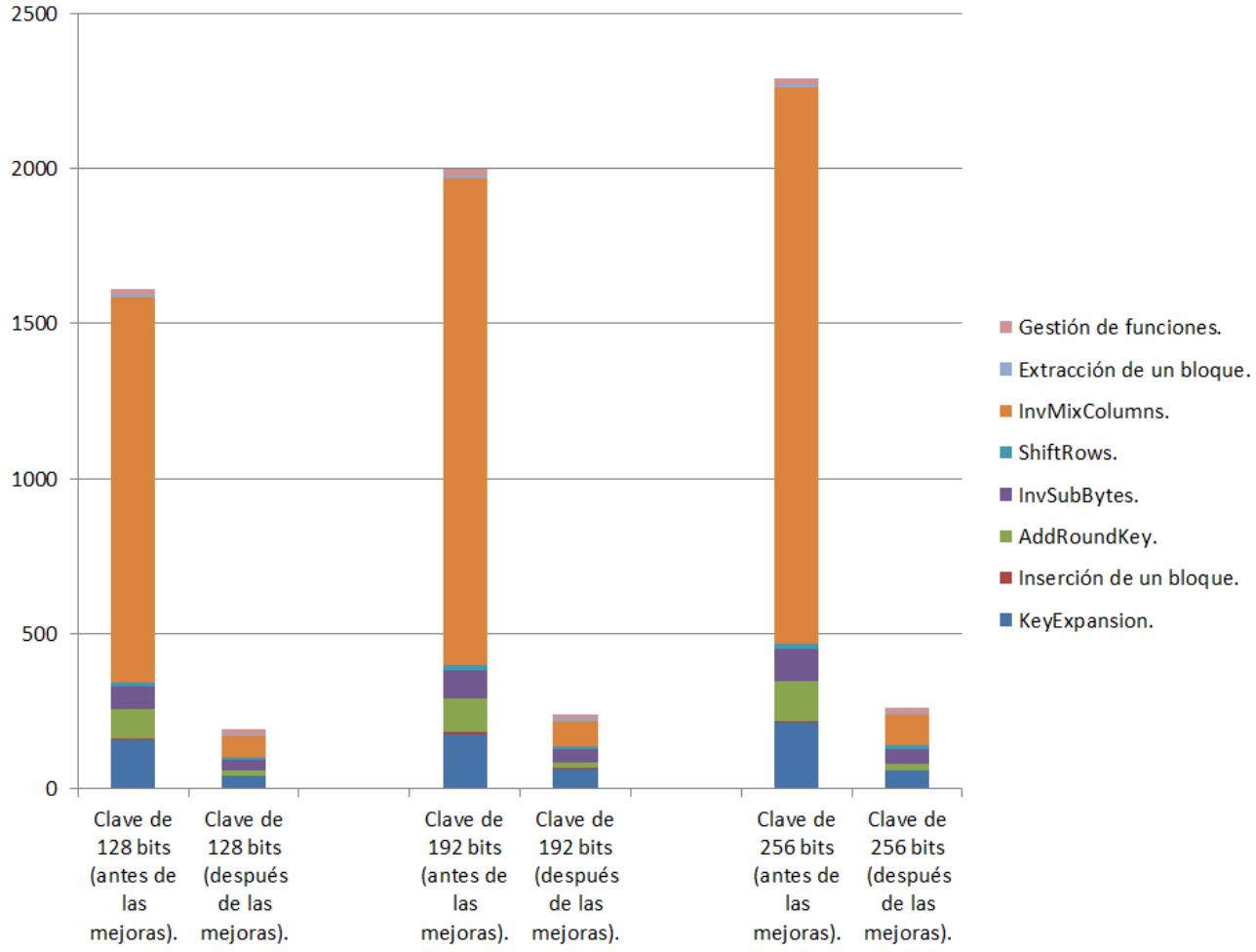

Figura 29. Rendimiento total de las mejoras descifrando un bloque.

En esta figura podemos ver el gran peso que tiene la función InvMixColumns así como el gran esfuerzo que se ha hecho para reducir su tiempo de proceso y el efecto que ello produce en la ejecución final.

Si se observan las cifras del tiempo total se puede ver como antes de las mejoras era un tiempo excesivo que podía imposibilitar realizar aplicaciones rápidas de descifrado.

No obstante para analizar un uso más común se han tomado también tiempos del descifrado de 10 bloques y se desglosan en la figura 30.

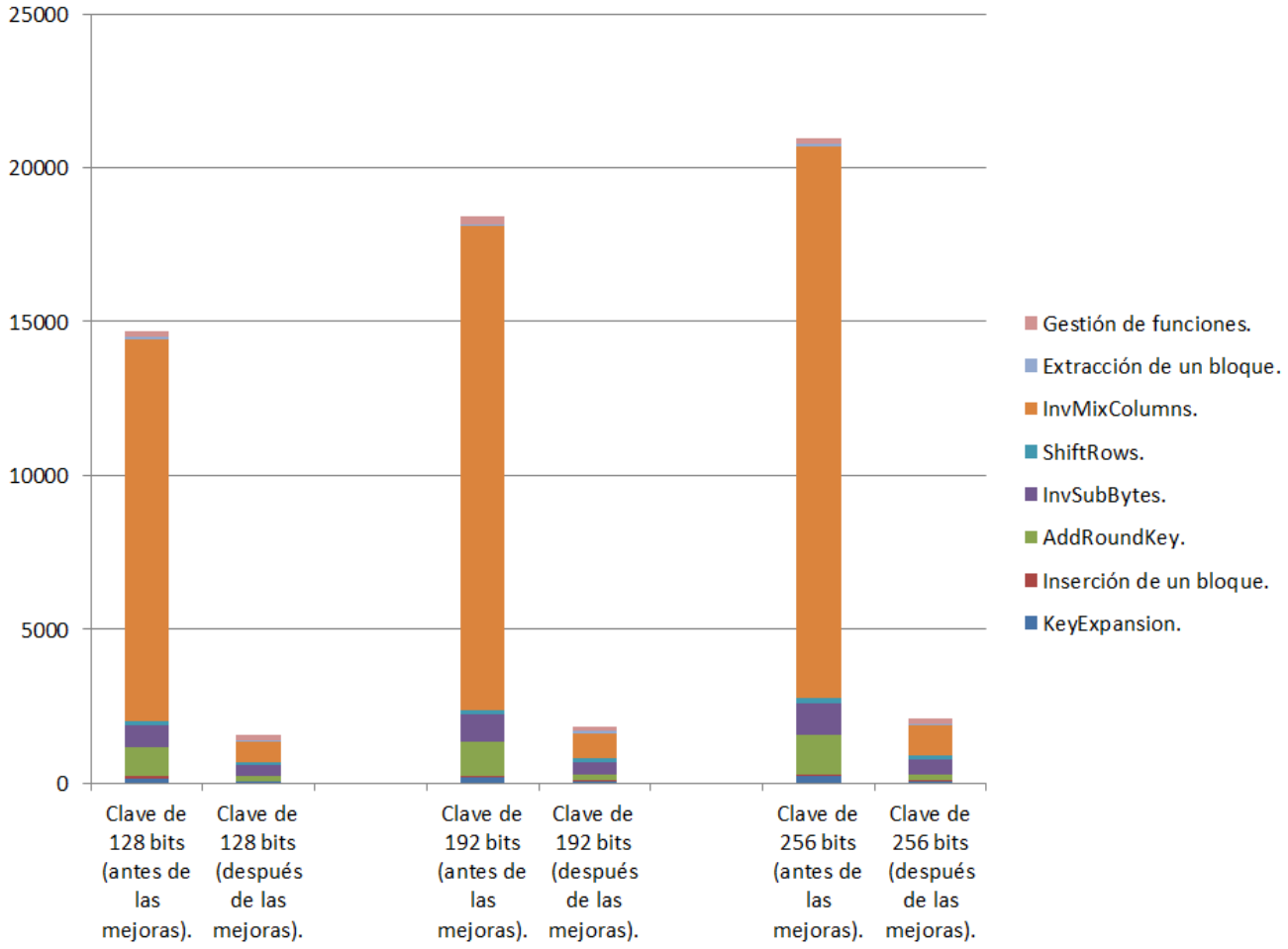

Figura 30. Rendimiento total de las mejoras descifrando 10 bloques.

Se puede apreciar que las mejoras han reducido enormemente el tiempo de proceso de la función InvMixColumns, y sumándolo a la reducción del resto de funciones se obtiene una reducción del 90% del tiempo empleado en descifrar 10 bloques de cifrado. Gracias a ello se alcanzan tiempos menores que un cuarto de milisegundo por bloque.

En la gráfica se aprecia la gran reducción de InvMixColumns. Esto es debido a que tras el análisis previo se trabajó mucho en esa función. No obstante no se trabajó solo en ella y de hecho, como ya se comentó anteriormente los tiempos del resto de funciones también han sido muy reducidos, solo que en esta gráfica la reducción de InvMixColumns se visualiza tan alta que el resto de reducciones pasan inadvertidas

La reducción total tiene efecto con claves de 128, 192 o 256 bits, o sea en todos los posibles tamaños de clave, y a mayor número de bloques a cifrar será mayor, puesto que la función KeyExpansion será ejecutada una sola vez y su mejora (75%) pesará menos que la de InvMixColumns (95%).

Por tanto se puede concluir diciendo que se ha reducido enormemente el tiempo de proceso del algoritmo para todos los posibles tamaños de clave, tanto en el cifrado como en el descifrado.

Estas reducciones se han realizado aprovechando las capacidades del microprocesador y las cualidades del algoritmo, reduciendo tiempos a lo largo de todo el algoritmo pero sobre todo en los puntos más pesados en cuanto a tiempo.

Además, gracias a haber reducido el número de variables y de operaciones, se ha minimizado la cantidad de memoria utilizada por el código y se ha reducido de 18.852 Bytes a 16.516 Bytes. Esto supone una reducción de más del 12% en la cantidad de memoria utilizada.

Resumiendo, se ha reducido el tiempo de cómputo y la memoria utilizada, por tanto se ha considerado como lo más oportuno mejorar la seguridad del algoritmo aprovechando la reducción conseguida en el tiempo de procesamiento. Por esto a continuación se aplicarán las mejoras de seguridad que son explicadas en el siguiente apartado.

4.3 *Mejoras de seguridad*

El algoritmo AES es considerado uno de los algoritmos más resistentes a ataques. No obstante la criptografía todavía no es totalmente segura y como consecuencia pueden aparecer agujeros de seguridad.

En nuestro algoritmo solamente se ha encontrado uno que merezca la pena comentar por su importancia y ha sido recientemente, a finales de Septiembre de 2011. Según [20] tres expertos realizaron un criptoanálisis por grafos bipartitos (consistente en dividir el problema en varios problemas menores) y encontraron un defecto que provoca que AES sea cuatro veces más débil. No obstante, incluso aprovechando este defecto un ataque resulta computacionalmente inviable, ya que se necesitarían más de 2000 millones de años para romper una clave de 128 bits mediante esta técnica y fuerza bruta.

Podemos ver como a pesar de la búsqueda de defectos que se está realizando, el algoritmo demuestra tener una gran seguridad. No obstante, los ataques de canal lateral son capaces de eludir la estructura interna del algoritmo y hacen viable el criptoanálisis.

Por ello en este proyecto se ha reforzado el algoritmo frente a este tipo de ataques añadiendo máscaras aleatorias a la matriz de estado.

4.3.1 Obtención de números aleatorios

La obtención de máscaras aleatorias exige que nuestro microprocesador cree números aleatorios que se puedan usar para calcular las máscaras.

Existen varias maneras de obtener la aleatoriedad.

4.3.1.1 Módulo de creación de números aleatorios

Algunos microprocesadores como el LPC3130 de la misma marca NXP tienen un módulo de creación de números aleatorios (Random Number Generator, RNG) basado en un par de anillos osciladores de reloj independientes que varían de un dispositivo a otro dependiendo de la fabricación del componente y de la temperatura del dispositivo.

Según el manual de usuario de estos microprocesadores [21] para utilizar este módulo solo bastaría con configurarlo correctamente en el inicio del programa mediante el

registro *POWERDOWN* y leer el contenido del registro de 32 bits *RANDOM_NUMBER* cada vez que quisiéramos obtener un número aleatorio.

Esta sería manera más oportuna de obtener números aleatorios, no obstante, como el módulo de generación de números aleatorios no está presente en todos los microcontroladores, se decidió usar un microcontrolador más genérico e implementar la obtención de números aleatorios en él ya que es una solución mucho más amplia y se podría aplicar en múltiples plataformas.

4.3.1.2 Funciones *srand(semilla)* y *rand()*

Existe una librería en C llamada *stdlib.h* que incluye los métodos *rand()* y *srand()* que se encargan de generar números aleatorios. Su funcionamiento es el siguiente:

- int rand(void);
  La función rand devuelve un número entero de una secuencia de números pseudo-aleatorios con un periodo de 32 bits como mínimo.
- void srand(unsigned int semilla);
  Usa el argumento como una semilla para crear una secuencia nueva de números pseudo-aleatorios que podrán ser retornados por llamadas posteriores a rand. Si srand es entonces llamada con el mismo valor semilla, la secuencia de números pseudo-aleatorios será repetida. Si rand es llamada antes de que se hayan hecho cualquier llamada a srand, la misma secuencia será generada.

Por tanto si obtenemos una semilla aleatoria y la cargamos en *srand(semilla)* obtendremos números de 32 bits aleatorios en las sucesivas llamadas a *rand()*.

El uso de estas funciones implica que nuestro programa utilice 408 bytes más de memoria y que cada llamada a la función rand() suponga un incremento de 0.133 microsegundos en el tiempo de ejecución del programa lo cual significan 0.333 microsegundos en una operación de cifrado o descifrado. Como estos datos son perfectamente asumibles, se ha tomado ésta como la manera de obtener números aleatorios.

No obstante esta forma de obtenerlos parte de una semilla que tiene que ser prácticamente aleatoria.

4.3.1.3 Obtención de una semilla aleatoria

Dentro de un microprocesador podría decirse que todos los datos que podemos obtener son deterministas por lo que obtener un dato que nos pueda servir de semilla aleatoria puede parecer complicado.

La única manera de resolver esta complejidad es introduciendo parámetros no deterministas que vengan de dentro o fuera del microprocesador como pueden ser la temperatura interna o tiempos marcados por el usuario entre otros.

Conviene además que no se tome solo uno por si acaso dejara de ser determinista por alguna razón.

En nuestro caso, se atendió a las funcionalidades que incluía el microprocesador y se han tomado dos parámetros no deterministas:

- Un tiempo marcado por el usuario sin que este lo sepa.

  Para ello se ha utilizado el timer0 configurado con un preescalado de 1 para que la frecuencia sea lo mayor posible y no sea posible prever el tiempo transcurrido. El timer0 comienza a contar al inicio del programa y se accede a su valor cuando se

generan las máscaras, entre la recepción de la clave y la petición del bloque de cifrado. De esta manera el tiempo transcurrido es variable de una vez a otra por lo que aportará un valor distinto cada vez. Además, como el usuario no conoce el momento exacto en el que se inicia la cuenta ni el momento en el que se toma el dato y como la frecuencia del temporizador es de 60 megahertzios, la posibilidad de saber cual va a ser el valor obtenido es prácticamente nula.

- Una medida del ruido ambiental.

  En este caso se utiliza el conversor analógico digital (*ADC*) para transformar una medida analógica del exterior (puerto 0.30).

  El *ADC* toma valores analógicos entre 0 y 3 voltios y los cuantifica devolviendo un valor digital de 10 bits. Por tanto, el paso de cuantificación será de 2,93 mV. Esto quiere decir que cada 2,93 mV el ADC devolverá un valor distinto.

  Para hacer que este valor sea más variable se realizarán 8 tomas en distintos momentos de los 4 bits menos significativos (Más variables) de la salida amplificada de un micrófono y al concatenar estas medidas se formará una palabra de 32 bits aleatoria. Al coger los cuatro últimos bits la relación entre dos medidas distintas es nula puesto que se deshecha la componente de continua y se toma la parte más variable.

  El micrófono utilizado es un micrófono electrect, que da una salida con valores de tensión muy pequeños. Para adaptarla al rango de valores que recibe el *ADC* se ha diseñado un circuito de amplificación.

  En la figura 31 se puede observar el esquema del circuito y su implementación junto con la placa de desarrollo del LPC2132.

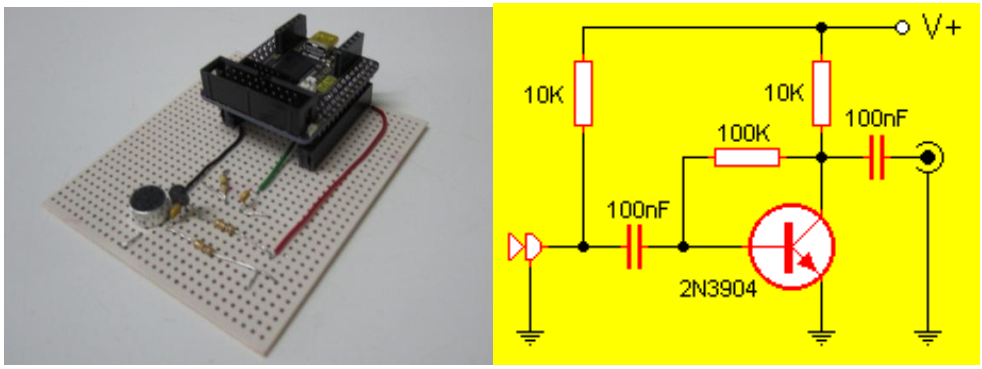

Figura 31. Circuito de amplificación del micrófono. [22]

El circuito se conecta a las salidas de tensión (3,3V) y de tierra del microcontrolador y tras haber sido probado después de ser realizado, tiene una salida con valores entre 0 y 1 voltio, más que suficiente para aportar un valor aleatorio en los últimos cuatro bits del CAD.

En caso de que el microprocesador formara parte de un dispositivo móvil, el ruido ambiental podría tomarse a través del micrófono del terminal o incluso tomar un valor de cantidad de luz que le llega a la cámara.

A estos parámetros se le suma además mediante una operación XOR un valor constante que podría cambiarse para cada aplicación, por lo que la raíz de la semilla aleatoria es la suma de este valor fijo, una medida de un tiempo aleatorio y otra del ruido acústico.

Resumiendo se puede decir que utilizando esta semilla aleatoria como parámetro en la función *srand(semilla)* y llamando a la función *rand()* se pueden obtener máscaras aleatorias aunque todavía no se ha especificado como usar estas máscaras.

4.3.2 Uso de máscaras aleatorias

Para proteger el algoritmo frente a ataques de canal lateral, la solución más oportuna consiste en añadir distintas máscaras aleatorias a la matriz de estado a lo largo del proceso de cifrado o descifrado. De esta manera el consumo de potencia no guarda relación con el mensaje a cifrar o descifrar ni con la clave de cifrado.

No obstante, la manera de aplicar estas máscaras no debe ser tomada a la ligera, si no que ha de ser correctamente estudiada para no comprometer lo más mínimo la seguridad del proceso. Tras estudiar distintas formas de añadir máscaras, debido a su sencillez y efectividad, se tomó como principal referencia la propuesta realizada por Stefan Mangard, Elisabeth Oswald y Thomas Popp en su libro *Power Analysis Attacks: Revealing the Secrets of Smart Cards* [23] y se adaptó al funcionamiento en palabras de 32 bits en lugar de 8.

4.3.2.1 Máscaras utilizadas

En este diseño se ha utilizado enmascarado booleano[5]. Se basa en enmascarar las transformaciones de ronda aunque no se olvida enmascarar la generación de claves. Por tanto se generan máscaras tanto para el texto a cifrar como para la primera clave de ronda.

En total se utilizan 10 máscaras distintas, 5 de entrada ($m, m_1, m_2, m_3\ y\ m_4$) y 5 de salida ($m', m'_1, m'_2, m'_3\ y\ m'_4$). Cada una de ellas de un byte de tamaño.

Las máscaras $m, m_1, m_2, m_3, m_4 y\ m'$ son obtenidas a partir de números completamente aleatorios, mientras que $m'_1, m'_2, m'_3\ y\ m'_4$ al ser las máscaras de salida de la operación MixColumns se obtendrán realizando esta operación a las máscaras $m_1, m_2, m_3\ y\ m_4$.

Conviene notar que las máscaras utilizadas son de un tamaño de 8 bits (1 byte) y el programa trabaja con palabras de 32 bits de tamaño. Por esto, cuando se utilice una misma máscara en todos los bytes se utilizará una palabra de 32 bits con el mismo byte concatenado 4 veces. Cuando los bytes de una palabra lleven distintas máscaras, se concatenarán las máscaras en el orden oportuno.

A continuación describiremos como son utilizadas en las cuatro principales operaciones del proceso.

4.3.2.1.1 AddRoundKey

Al inicio de la operación AddRoundKey los bytes de las claves de ronda (*k*) estarán enmascarados con la máscara *m*. Por tanto, al realizar la operación se enmascararán automáticamente los bytes *d* de la matriz de estado siguiendo esta manera:

$$d \oplus (k \oplus m) = (d \oplus k) \oplus m$$

[5] Enmascarado booleano: Se realiza aplicando máscaras mediante operaciones OR-exclusivas en lugar de aritméticas (Enmascarado aritmético). De esta manera el valor enmascarado se puede obtener según esta ecuación: $v_m = v \oplus m$ siendo *v* el valor a enmascarar y *m* la máscara.

Además, tanto la matriz de estado como la subclave de ronda estarán enmascaradas con m'_1, m'_2, m'_3 y m'_4. De esta manera estas máscaras se anularán y los bytes de la matriz de estado pasarán de estar enmascarados con las máscaras m'_1, m'_2, m'_3 y m'_4 a estarlo con m.

4.3.2.1.2 SubBytes

La operación SubBytes tiene a m y a m' como máscaras de entrada y salida respectivamente.

Al ser la única operación no lineal de AES, esta operación se implementaba mediante una tabla.

Por tanto ahora habrá que crear otra tabla que realice esta operación y a la vez realice el cambio de máscara. Esta tabla deberá haber sido computada antes de comenzar el encriptado y para realizar el cambio de máscara tendrá que cumplir que

$$S_m(x \oplus m) = S(x) \oplus m'$$

4.3.2.1.3 ShiftRows

La operación ShiftRows rota las distintas filas un determinado número de bytes. Como en este momento todos los bytes están enmascarados con la misma máscara, el enmascarado no requerirá modificar esta función.

4.3.2.1.4 MixColumns

Con MixColumns hay que tener un especial cuidado ya que al mezclar los bytes de las distintas filas entre sí, podrían anularse las máscaras de estas filas si fueran iguales. Por ello antes de esta operación conviene cambiar las máscaras de la matriz de estado de m'a m_1, m_2, m_3 y m_4, para que cada fila tenga una máscara distinta. Esta nueva operación se llamará **Remasking**.

Tras esto bastará con aplicar la función MixColumns y la matriz de estado finalizará con las máscaras m'_1, m'_2, m'_3 y m'_4.

4.3.2.2 Orden de aplicación

Por tanto, tras aplicar las máscaras tal y como se comenta en el apartado 4.3.2.1 anterior los cambios que va sufriendo la matriz de estado y las máscaras por las que va pasando a lo largo del cifrado quedan representados en la figura 32.

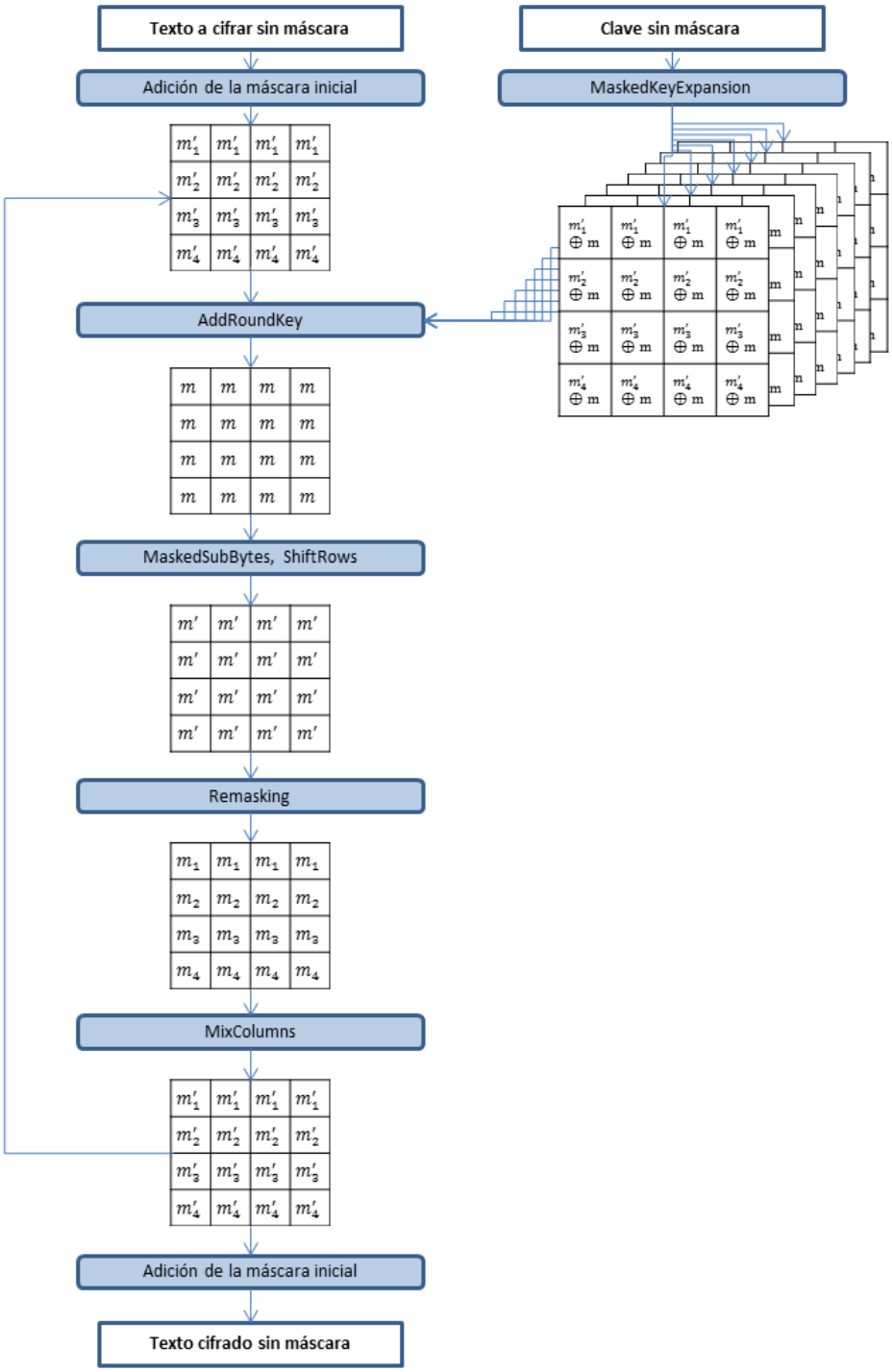

Figura 32. Evolución de máscaras.

En ella podemos ver como tras realizar la operación MixColumns la matriz de estado queda con las máscaras necesarias para realizar la operación AddRoundKey, lo cual evita realizar otra operación de reenmascarado y facilita los cálculos. Por tanto podemos decir que este método de trabajo está pensado para operaciones repetitivas o de rondas como es nuestra aplicación.

4.3.3 Implementación de las mejoras de seguridad

Hasta ahora se ha explicado como se construyen las máscaras y en que orden se aplican pero no se ha explicado que operaciones se encargan de ello ni el orden de las mismas dentro de todo el conjunto de la aplicación.

En este apartado se va a explicar en un primer momento como es el funcionamiento interno del programa y posteriormente cuales son los distintos archivos que lo componen junto con sus funciones y variables.

4.3.3.1 Funcionamiento del programa

En la figura 33 se muestra el diagrama de flujo del funcionamiento de AES con las mejoras de seguridad propuestas.

Es parecido a los diagramas de flujo anteriores pero en este caso se ha resumido cierta información para que se puedan apreciar mejor los cambios realizados. Las partes resumidas son la petición y espera de los parámetros de cifrado y la petición y espera de la clave entre otras.

El primer cambio que se puede ver es la llamada a iniMascaras(), que configura los periféricos necesarios para la obtención de máscaras (El *timer* y el *ADC*).

Tras esto se activa la interrupción del ADC para que éste comience a tomar muestras del ruido acústico.

Se puede apreciar también como tras recibir la clave de cifrado y calcular el número de rondas se ejecutan las funciones genMascaras() y maskSBox() encargadas de crear las máscaras a utilizar y de generar la tabla SBox enmascarada respectivamente.

En caso de que se vaya a descifrar datos se crea también la inversa de la tabla SBox enmascarada mediante la llamada a la función maskISBox().

Tras esto el funcionamiento es el mismo exceptuando la llamada a maskedKeyExpansion() en lugar de a keyExpansion() y la llamada a la función remasking() que aparece como novedad y se realiza después de shiftRows() o de invShiftRows().

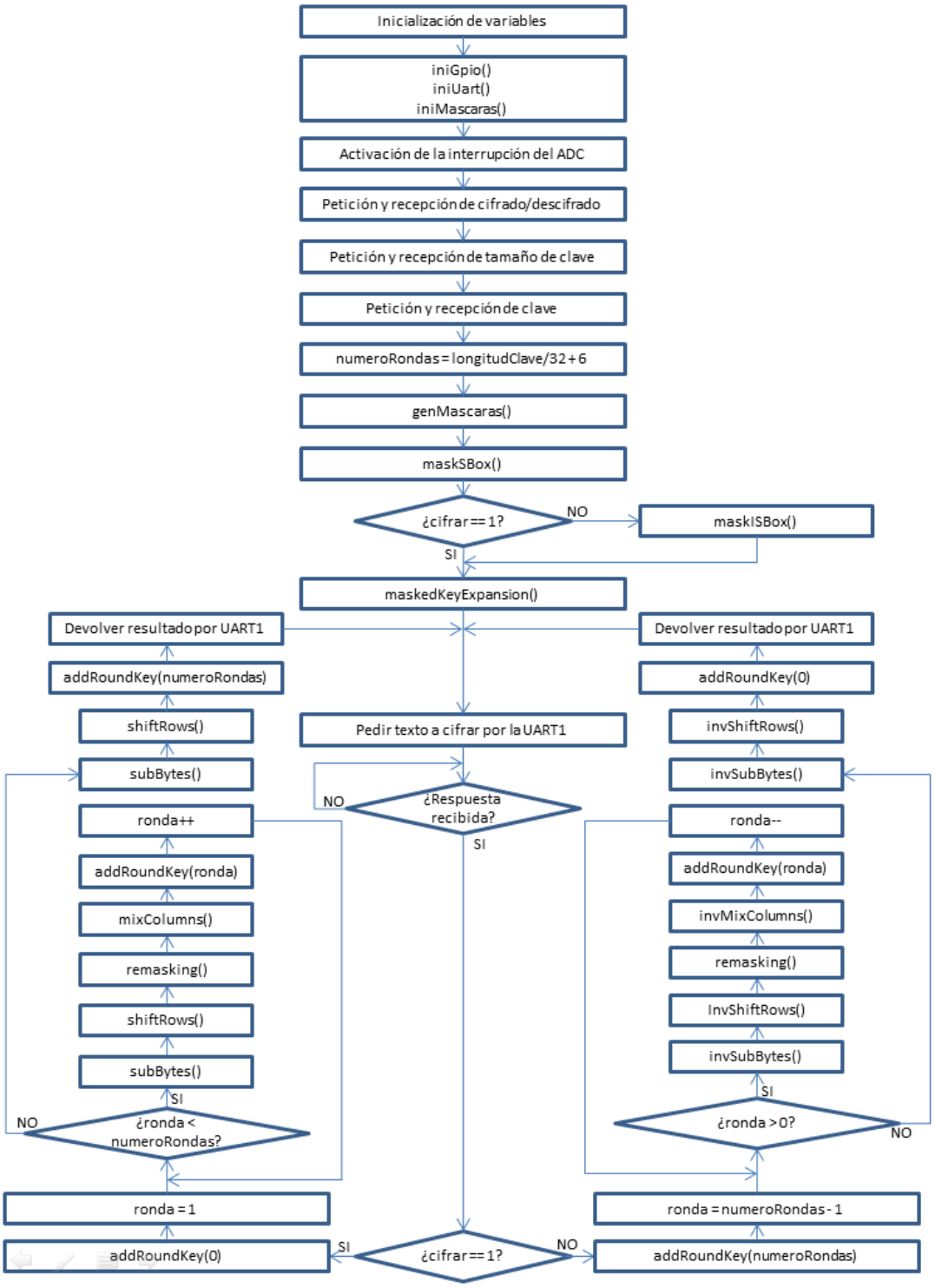

Figura 33. Diagrama de flujo del programa.

Además también podemos ver el diagrama de flujo de la interrupción que gestiona el conversor analógico digital en la figura 34.

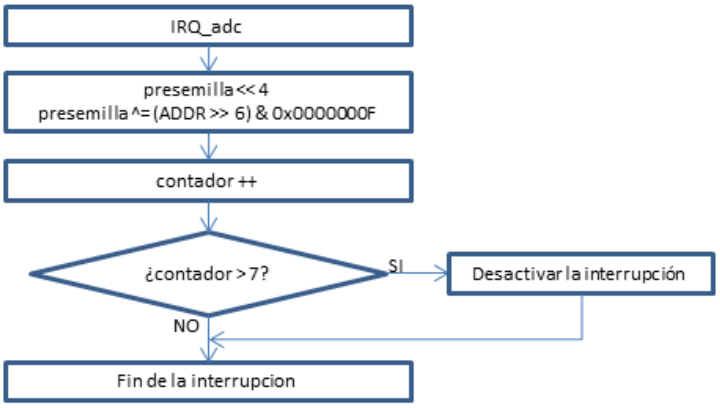

Figura 34. Interrupción del ADC.

En esta figura podemos ver como cada vez que salte la interrupción (Cada vez que se termine una conversión) se tomará el valor de los cuatro bits menos significativos (Los más variables) y se irán concatenando en la variable presemilla.

Una vez se haya hecho esto 8 veces se habrán tomado 32 bits aleatorios y se desactivará la interrupción.

4.3.3.2 Distribución del programa

En este caso el programa realizado se ha dividido en 5 archivos diferentes. Cada uno de estos archivos contiene funciones y variables clasificadas por su utilidad.

Los archivos del programa son AES.c, libreria_mascaras.c, libreria_uart1.c y main.h. y en la tabla 22 se indica que funciones y variables acoge cada uno de ellos. Además ofrece una breve descripción sobre la utilidad y el funcionamiento de algunas de las funciones o variables.

aes.c	Variables	int sbox[256]	Tabla S-box.
		int isbox[256]	Tabla S-box inversa.
		int maskedSbox[256]	Tabla S-box enmascarada.
		int maskedIsbox[256]	Tabla S-box inversa y enmascarada.
		unsigned int estado[4]	Array con las columnas de la matriz de estado.
		unsigned int TRoundKey[60]	Array con el conjunto de todas las subclaves de ronda.
		int Rcon[12]	Constantes Rcon.
	Funciones	setEstado(entrada[16])	Coloca los componentes del array de entrada en sus lugares dentro de la variable estado[4].
		getEstado()	Devuelve un puntero al array de la matriz de estado
		addRoundKey(ronda)	Realiza la operación AddRoundKey entre la matriz de estado y la correspondiente subclave de ronda depositando el resultado en la matriz de estado
		subBytes()	Estas operaciones se corresponden con las operaciones principales del algoritmo AES y sus inversas. Trabajan directamente sobre la matriz de estado y no necesitan ningún parámetro.
		invSubBytes()	
		shiftRows()	
		invShiftRows()	
		mixColumns()	
		invMixColumns()	

libreria_mascaras.c	Variables	int parametroFijo	Parámetro que el programador puede modificar para añadir algo más de aleatoriedad a la semilla.
		int presemilla	Almacena los bits aleatorios obtenidos del ADC.
		unsigned int mascaras[10]	Almacena las diez máscaras aleatorias.
	Funciones	iniMascaras	Inicializa el ADC y el timer0 para capturar datos aleatorios
		genMascaras	Obtiene el valor de tiempo, establece la semilla y genera las diez máscaras de cifrado, almacenándolas de la manera oportuna.
		maskSbox	Genera la tabla maskedSbox.
		maskISbox	Genera la tabla maskedISbox.
		maskedKeyExpansion	Crea todas las subclaves de ronda enmascaradas a partir del contenido de la variable clave de main.c
		remasking	Cambia la máscara de la matriz de estado.
libreria_uart1.c	Variables	caracteresAEnviar[50]	Buffer de envío de datos por el puerto serie de 50 caracteres de capacidad.
	Funciones	iniUart1()	
		enviarCadena_uart1(cadena)	
		enviarSaltoDeCarro_uart1()	
main.c	Variables	cifrar	1 – cifrar, 0 – descifrar.
		longitudClave	
		clave[32]	Array que guarda la clave de cifrado. Su tamaño es el mayor posible, en caso de que la clave no sea de la mayor longitud (256 bits) las últimas posiciones del array se ignorarán.
		claveRecibida[64]	Almacena los caracteres recibidos como clave.
		palabraRecibida[32]	Almacena los caracteres recibidos como bloque a cifrar.
		bloque[16]	Array que guarda el bloque a cifrar tras averiguar el significado de los caracteres recibidos.
		palabraProcesada[16]	Guarda el bloque resultante del proceso de cifrado/descifrado.
		numeroRondas	
		nuevaPalabra	Indica cuando se ha recibido todo un bloque por el puerto serie.
	Funciones	fijaBloque()	Averigua el valor real de los caracteres ASCII recibidos como bloque de cifrado y lo almacena en la variable bloque.
		fijaClave()	Averigua el valor real de los caracteres ASCII recibidos como clave y lo almacena en la variable clave.
		leeCifra(caracter)	Devuelve el valor real de un carácter ASCII
		leeChar(valor)	Devuelve el carácter ASCII de un valor numérico
		main()	Método main del programa. Comienza a partir de él.

Tabla 22. Funciones y variables de la versión con mejoras de seguridad.

En conjunto, estos cinco archivos permiten ejecutar el algoritmo AES con las mejoras de rendimiento del apartado 4.2. y posibilitan que se utilicen máscaras de cifrado para mejorar la seguridad tal y como se ha descrito a lo largo de este apartado.

4.3.4 Rendimiento tras la aplicación de las mejoras

Tras aplicar las citadas mejoras de seguridad lo óptimo sería realizar un análisis de efectividad de las mismas.

La mejor manera de realizar este análisis sería realizando un ataque de canal lateral (Por potencia consumida o por campos magnéticos emitidos) al programa en funcionamiento dentro del microprocesador.

No obstante la protección frente a ese tipo de ataques mediante máscaras aleatorias es una técnica ampliamente contrastada que se utiliza en multitud de aplicaciones y cuya efectividad ya ha sido demostrada en [23] entre otros.

Además, este tipo de ataques son muy costosos y su implementación conlleva materiales y esfuerzos que se escapan de este proyecto, tanto por materia como por tiempo.

Por ello se adoptarán los estudios ya realizados como válidos y en este apartado solamente se analizará el coste computacional que tiene la adición de las máscaras.

El aumento de tiempo se traduce en los tiempos de proceso que aparecen en la tabla 23. Anteriormente se explicó que el tiempo de una sola ronda no es el más representativo, por ello en este caso se han tomado tiempos y se han realizado gráficas a partir de los tiempos correspondientes a cada función durante diez rondas de cifrado AES con máscaras.

	Cifrado de 10 bloques			**Descifrado de 10 bloques**		
	128 bits de clave	192 bits de clave	256 bits de clave	128 bits de clave	192 bits de clave	256 bits de clave
Tiempo sin mejora de seguridad.	1.283ms	1.509ms	1.707ms	1.553ms	1.839ms	2.097ms
Tiempo tras mejora de seguridad.	1.543ms	1.795ms	2.023ms	1.877ms	2.197ms	2.482ms
Aumento de tiempo.	0.260ms (**20.26%**)	0.286ms (**18.95%**)	0.316ms (**18.51%**)	0.344ms (**22.15%**)	0.358ms (**19.46%**)	0.385ms (**18.36%**)

Tabla 23. Tiempo de proceso tras mejoras de seguridad.

El coste computacional de aumentar la seguridad frente a ataques de canal lateral está por tanto alrededor del 20%.

Por supuesto que es una cantidad notable, no obstante ya se había reducido considerablemente el tiempo de proceso por lo que un 20% es una cantidad perfectamente asumible, sobre todo teniendo en cuenta la seguridad frente a ataques de canal lateral obtenida.

Además, el hecho de añadir estos valores implica un aumento muy ligero de la memoria utilizada, de 416 bytes para ser más exactos. Por tanto la memoria necesaria pasará de 16516 a 16932 bytes, un aumento de tan solo el 2.52% que merece la pena asumir.

Para comprender cuál es el punto del que viene el aumento de tiempo se adjuntan las figuras 35 y 36.

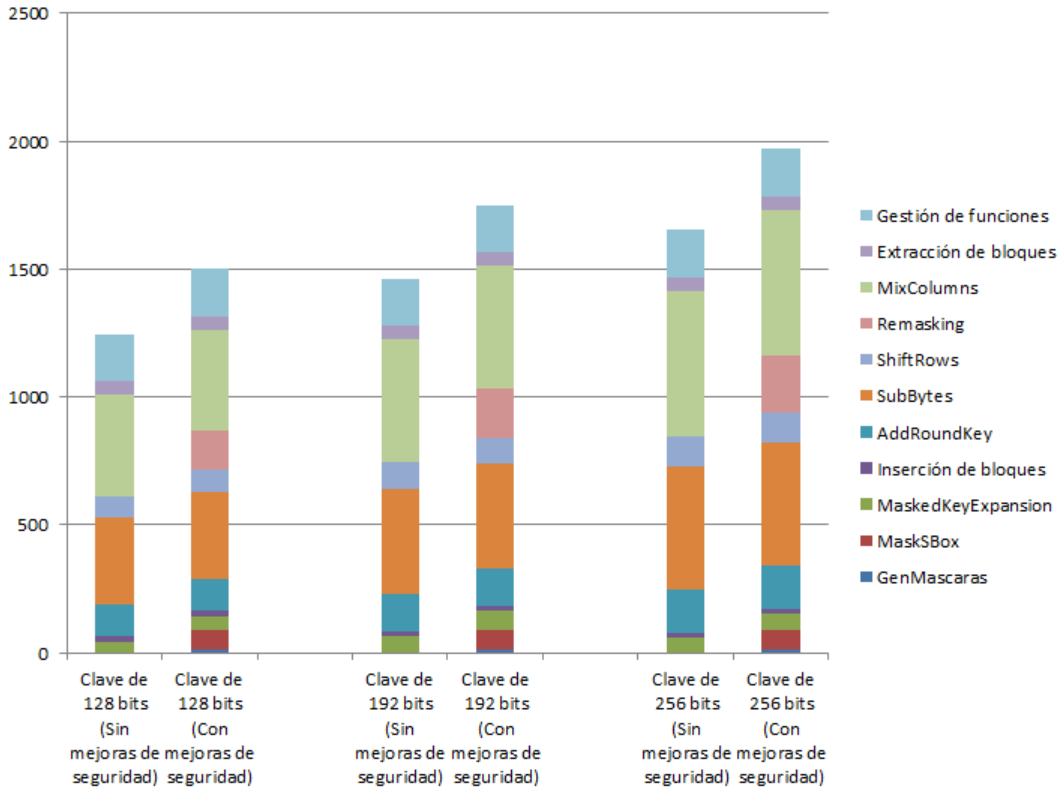

Figura 35. Rendimiento tras mejoras de seguridad (Cifrado).

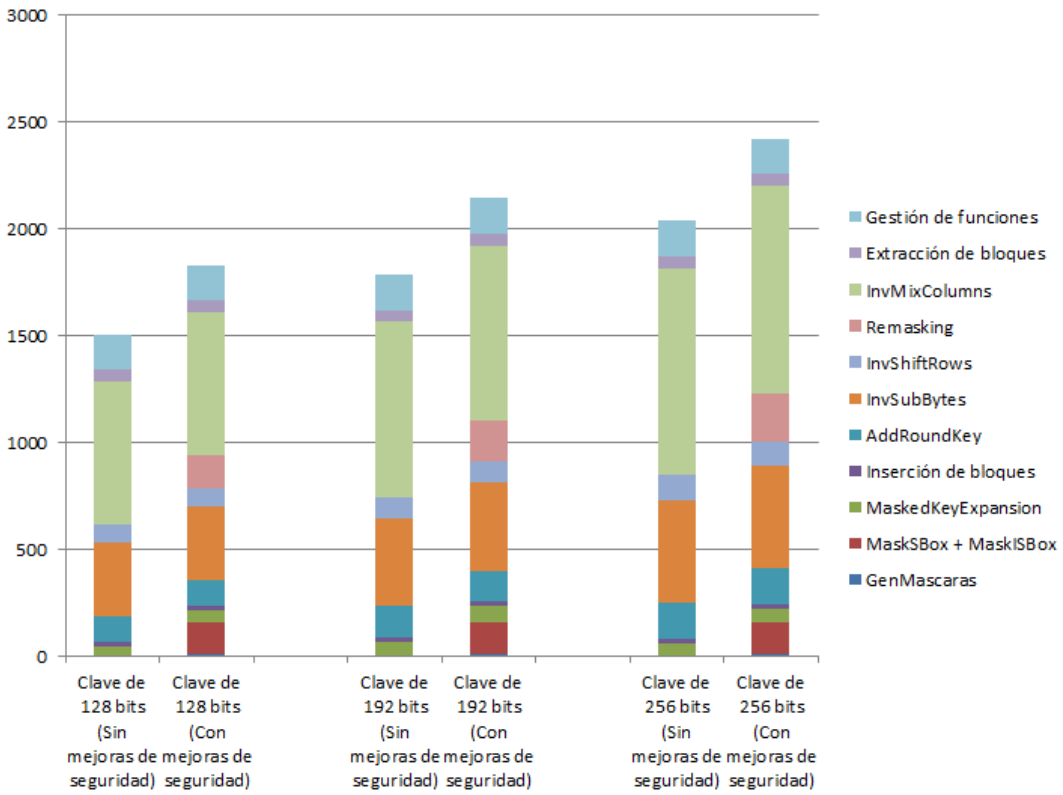

Figura 36. Rendimiento tras mejoras de seguridad (Descifrado).

En ellas se puede ver que hay un incremento de tiempo de proceso debido principalmente a la función remasking y a la generación de la tabla S-Box enmascarada y a la de su inversa.

Se puede ver que la generación de las máscaras y el enmascaramiento de la generación de las subclaves ocupan muy poco tiempo, al contrario que la generación de las tablas S-Box o la función remasking.

La generación de las tablas enmascaradas se efectuaría una sola vez, por lo que cuantos más bloques de cifrado se realizaran menos peso tendría. La mayoría de las aplicaciones cifran una cantidad de bloques mucho mayor que diez por lo que en la mayor parte de los casos el tiempo de generación de estas tablas será ínfimo en comparación del tiempo total.

Si se analizan esos casos los resultados obtenidos pueden llegar a acercarse a los valores de la siguiente tabla.

	Cifrado con 100 o más bloques			**Descifrado con 100 o más bloques**		
	Clave de 128 bits	Clave de 192 bits	Clave de 256 bits	Clave de 128 bits	Clave de 192 bits	Clave de 256 bits
Aumento de tiempo.	13.66%	14.21%	14.62%	11.23%	12.55%	14.98%

Tabla 24. Aumento de tiempo en cifrados largos.

La mayoría de aplicaciones cifran más de cien bloques por lo que se puede concluir diciendo que en la mayoría de las aplicaciones esta mejora de seguridad solo supone un aumento del coste computacional del **15%** y un aumento insignificante en la memoria utilizada.

Capítulo 5

Conclusiones

Tras haber planteado una serie de objetivos y haber realizado acciones para llegar a ellos, en este capítulo se analizarán los resultados alcanzados.

El primer objetivo era implantar el algoritmo AES en el microprocesador siguiendo fielmente las especificaciones de Daemen y Rijmen [18]. Gracias a la comunicación por el puerto serie y al uso de interrupciones entre otras herramientas se ha integrado AES en el microprocesador, y se ha conseguido que funcione, tanto para cifrar como para descifrar, con cualquier tamaño de clave (128, 192 y 256 bits).

Para saber que opción de funcionamiento es la deseada se ha creado un menú de selección que es ejecutado antes del cifrado.

Una vez probado el correcto funcionamiento del algoritmo se trató de mejorar sus tiempos de funcionamiento. Para ello se realizó un análisis de tiempos para ver cuales son los puntos que aportan más retardo al algoritmo y un minucioso análisis del funcionamiento de las funciones del algoritmo para ver como podrían ser simplificadas. Mediante estos dos análisis se descubrió que los puntos en los que había que centrarse eran las funciones MixColumns e InvMixColumns.

No obstante se ha reducido el tiempo de proceso de todas las funciones del algoritmo gracias a la ejecución de operaciones con palabras de 32 bits en lugar de 8 y gracias a técnicas de loop unrolling entre otras.

La ejecución de operaciones con palabras de 32 bits en lugar de 8 bits merece una mención especial ya que es una adaptación del algoritmo a una gran cantidad de microprocesadores y con ella teóricamente basta con ejecutar una operación donde antes se necesitaban cuatro.

Gracias a esto se obtienen unos resultados excelentes, llegando a reducirse el tiempo del cifrado en más de un 60% y el de descifrado en un 90% consiguiendo además una reducción del 12% en la cantidad de memoria utilizada.

Tras haber aligerado sustancialmente el tiempo de ejecución se decidió hacer el algoritmo más robusto frente a ataques de canal lateral por consumo de potencia mediante la adición de máscaras.

Para ello se generó código encargado de obtener números aleatorios basados en una semilla formada por elementos indeterministas como el tiempo que el usuario tarda en introducir los parámetros de cifrado o el ruido de ambiente.

Para obtener el ruido ambiental se creó un circuito electrónico que obtiene un valor analógico del ruido acústico y lo envía al CAD del microprocesador para que obtenga los bits correspondientes.

De esta manera se han conseguido secuencias totalmente aleatorias, las cuales han permitido obtener máscaras aleatorias, que han sido utilizadas para enmascarar de una manera segura el algoritmo y protegerlo frente a ataques de canal lateral.

Este aumento de seguridad supone un insignificante aumento de la memoria utilizada (2.52%) y un aumento del tiempo de ejecución que en aplicaciones reales puede estar entre el 12% y el 20% respecto al tiempo tras las mejoras de rendimiento.

Con el objetivo de analizar los logros obtenidos y los costes de las mejoras de seguridad en la tabla 25 muestran los resultados obtenidos con las distntas mejoras implementadas. En esta tabla se pueden ver los tiempos de proceso de 10 rondas de cifrado o descifrado en cada una de las distintas versiones del software creadas y el porcentaje que dichas cantidades representan respecto al tiempo original.

	Cifrado de 10 bloques.			**Descifrado de 10 bloques.**		
	Clave de 128 bits	Clave de 192 bits	Clave de 256 bits	Clave de 128 bits	Clave de 192 bits	Clave de 256 bits
Tiempo original.	3.188ms	3.756ms	4.339ms	14.702ms	18.409ms	20.979ms
Tiempo tras mejoras de eficiencia. (% del original)	1.283ms **(40.24%)**	1.509ms **(40.18%)**	1.707ms **(39.34%)**	1.553ms **(10.56%)**	1.839ms **(9.99%)**	2.097ms **(10%)**
Tiempo tras mejora de seguridad. (% del original)	1.543ms **(48.4%)**	1.795ms **(47.79%)**	2.023ms **(46.62%)**	1.877ms **(12.77%)**	2.197ms **(11.93%)**	2.482ms **(11.83%)**

Tabla 25. Tiempos de las distintas versiones.

Estos valores son representados en la figura 37, de manera que será más útil para explicar los resultados. En ella aparecen los tiempos de las distintas versiones en milisegundos para el cifrado o descifrado de 10 bloques.

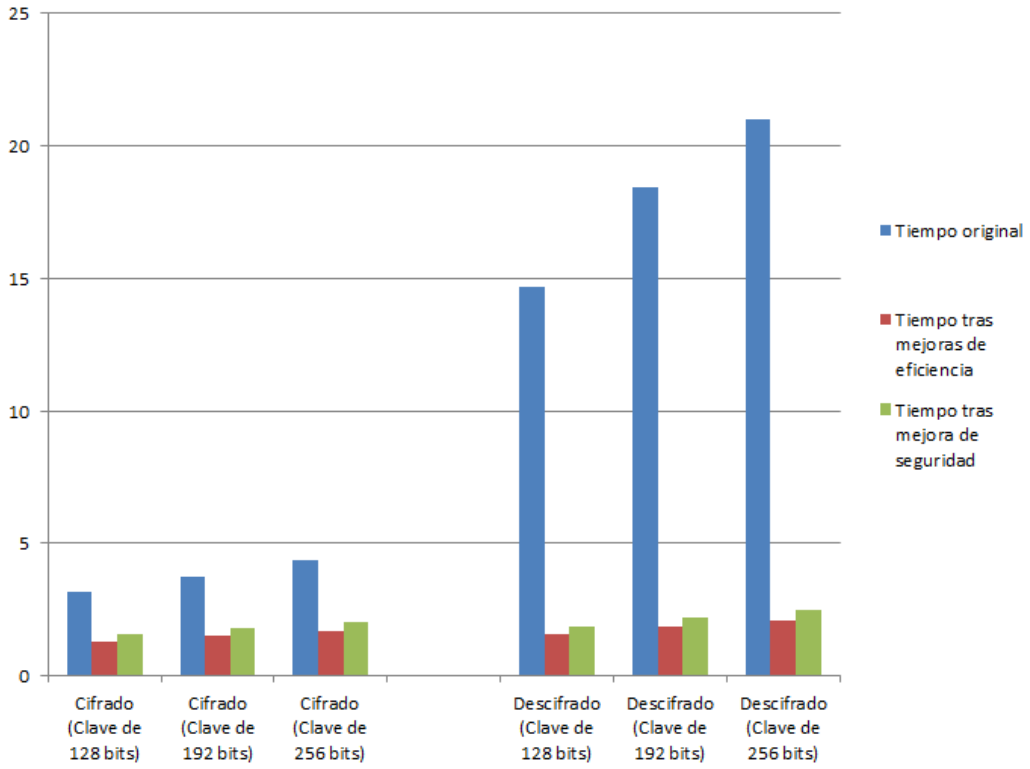

Figura 37. Comparativa de tiempos de las versiones de software.

Se observa como a mayor tamaño de clave, el tiempo de ejecución es mayor. Esto es debido a que una clave mayor implica más rondas para cifrar un solo bloque.

De todas formas, para cualquier tamaño de clave las mejoras de eficiencia han obtenido unos resultados muy buenos, sobre todo en el caso de descifrado.

La diferencia entre los tiempos originales del cifrado y el descifrado se debe principalmente a la función InvMixColumns, que era demasiado costosa al introducir un gran número de operaciones redundantes.

Tras las mejoras de eficiencia el descifrado tarda aproximadamente lo mismo que el cifrado.

Además se puede observar como el incremento de tiempo introducido por las mejoras de seguridad es muy pequeño en comparación con el tiempo original por lo que sin lugar a dudas ha merecido la pena ya que se ha ganado en resistencia a ataques de canal lateral y el tiempo de proceso sigue siendo muy inferior al inicial.

Capítulo 6

Trabajo futuro

A lo largo de este proyecto se ha conseguido que el microprocesador LPC2132 ejecute AES de una manera eficiente y con seguridad frente a ataques de canal lateral por potencia consumida.

Por tanto, los objetivos planteados han sido cumplimentados perfectamente. No obstante, se puede seguir trabajando en este proyecto para ampliar sus funcionalidades o mejorar el algoritmo.

Por tanto se proponen distintas maneras de continuar los trabajos realizados en este proyecto.

6.1 *Prueba de la resistencia frente a ataques de canal lateral*

Si se quisiera seguir trabajando en la parte de mejora del algoritmo, se podría probar de forma experimental la resistencia a los ataques de canal lateral por consumo de potencia antes y después de las mejoras de seguridad.

Para ello habría que tomar muestras de la tensión en distintos puntos del microcontrolador durante la operación AddRoundKey.

Con el fin de facilitar ese estudio, se ha desarrollado un conjunto de códigos iguales a los desarrollados exceptuando que en ellos el puerto 0.21 toma el valor 1 durante dicha operación y 0 en el resto de instantes con el fin de avisar al osciloscopio para que

comience o deje de medir. Estos códigos se encuentran en la carpeta “Códigos para prueba de ataque de canal lateral” del cd adjunto.

Además si al inicio de estos códigos se define la variable *_porPasos_* se habilitaría la ejecución por pasos condicionada a un posible pulsador que cambie el valor del puerto 0.20 entre cero y uno. De esta manera se podría iniciar o parar la ejecución del código a voluntad del usuario dando mucha flexibilidad al análisis.

6.2 *Creación de una aplicación encriptadora de archivos*

Si por el contrario se optara por trabajar añadiendo funcionalidades se propone crear una aplicación Java que interactúe con el microprocesador por puerto serie y que vaya intercambiando distintos bloques de cifrado con el microprocesador para encriptar o desencriptar archivos completos en lugar de bloques sencillos.

De esta manera se podría utilizar la portabilidad de Java para disponer de un cifrador AES de archivos externo al ordenador que funcionara en cualquier plataforma.

6.3 *Desarrollo multiplataforma*

Se propone también la adaptación de las mejoras desarrolladas a microprocesadores de otras familias o incluso a otros tipos de plataformas.

6.4 *Adaptación de las mejoras obtenidas a cualquier cifrador AES*

Por último también se propone utilizar las técnicas desarrolladas con la intención de mejorar la eficiencia en cualquier aplicación que ejecute AES y de esta manera conseguir que gracias a lo desarrollado en este proyecto el cifrado o descifrado AES sea mucho más rápido y eficaz.

Capítulo 7

Presupuesto

En este capitulo se van a desarrollar todos los costes que han supuesto realizar este proyecto, detallando cada uno de ellos para llegar al presupuesto total del proyecto.

7.1 *Planificación*

En primer lugar, se va a desarrollar la planificación que se ha seguido para la elaboración de este proyecto. En concreto se van a explicar las distintas fases de las que consta el desarrollo de dicho proyecto, indicando la duración de cada una de ellas y por tanto calculando la duración total de todo el proyecto.

Las fases de las que consta el proyecto son:

Fase 0: Estudio de la ciencia de la criptografía.

- Estudio genérico del estado actual de la criptografía.
- Estudio exhaustivo del algoritmo AES.

Fase 1: Estudio del dispositivo.

- Estudio genérico de los distintos dispositivos.
- Estudio exhaustivo de la placa ARM7 LPC2132.
- Aprendizaje en la programación del microprocesador.

Fase 2: Implementación del algoritmo AES.

- Programación del algoritmo dentro del microprocesador.
- Pruebas de funcionamiento.

Fase 3: Mejoras de eficiencia.

- Mediciones del funcionamiento del programa.
- Desarrollo de mejoras atendiendo a características del algoritmo.
- Desarrollo de mejoras atendiendo a características del programa y del microprocesador.
- Implementación de mejoras en el programa
- Pruebas de funcionamiento
- Medidas de eficiencia tras las mejoras.

Fase 4: Mejoras de seguridad.

- Estudio de mejoras de seguridad criptográfica.
- Implementación de mejoras de seguridad.
- Pruebas de funcionamiento.
- Medidas de rendimiento tras las mejoras de seguridad

Fase 5: Realización de la memoria y presentación de dicho proyecto.

- Realización de la memoria.
- Realización y ensayos de la presentación.

En la tabla 26 se muestra la duración de cada fase del proyecto así como su fecha de inicio y de fin. Tanto el día de fecha de inicio como el día de fecha de fin están incluidos en el proceso de elaboración de este proyecto. En este cálculo no se han contado los días festivos ni los días de vacaciones ni los fines de semana.

	Fecha inicio	Fecha fin	Duración
Fase 0	14/12/2011	29/12/2012	12 días
Fase 1	02/01/2012	13/01/2012	9 días
Fase 2	16/01/2012	07/02/2012	16 días
Fase 3	08/02/2012	13/03/2012	21 días
Fase 4	14/03/2012	06/03/2012	18 días
Fase 5	09/04/2012	21/04/2012	15 días

Tabla 26. Periodos del proyecto.

La duración total del proyecto es de 91 días. Si la media de dedicación a este proyecto ha sido de 6 horas diarias, entonces el número total de horas para este proyecto es de 546 horas.

7.2 *Presupuesto*

En el presupuesto se van a detallar los costes en primer lugar del personal utilizado, a continuación del hardware que se necesita y por último del coste del software necesitado.

7.2.1 Coste del personal

A continuación se va a detallar el coste del personal que es necesario para llevar acabo el proyecto. Estos roles han sido desarrollados por el autor de este proyecto.

	Rol	Coste (€/hora)	Duración	Coste
Fase 0	Analista	40	72 horas	2880 €
Fase 1	Analista	40	54 horas	2160 €
Fase 2	Programador	20	86 horas	1720 €
	Ingeniero de pruebas	30	10 horas	300 €
Fase 3	Analista	40	40 horas	1600 €
	Programador	20	76 horas	1520 €
	Ingeniero de pruebas	30	10 horas	300 €
Fase 4	Analista	40	36 horas	1440 €
	Programador	20	62 horas	1240 €
	Ingeniero de pruebas	30	10 horas	300 €
Fase 5	Analista	40	90 horas	3600 €
TOTAL			**546 horas**	**17060 €**

Tabla 27. Coste del personal.

La dirección de este proyecto ha sido llevada a cabo por el Dr. Luis Mengibar Pozo, que empleó aproximadamente 40 horas en llevar esta tarea. Puesto que sus honorarios ascienden a 40€/hora, el coste total de la dirección es 1600 €.

Por ello, el coste total del personal de este proyecto asciende a la cifra de 18660€.

7.2.2 Coste del hardware

Ahora se van a detallar los costes producidos por los elementos hardware que se han necesitado en este proyecto. Estos costes vienen detallados en la Tabla 28.

Concepto	Coste	% uso dedicado al proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable (*) (€)
Ordenador de sobremesa	700€	100	6	60 meses	70
Placa de desarrollo LPC2132	20€	100	6	60 meses	2
ULINK2 JTAG Debugger	300€	100	6	60 meses	30
Micrófono electrect y alimentación	6€	100	6	60 meses	0.6
TOTAL					**102.6 €**

Tabla 28. Coste del hardware.

(*) Fórmula de depreciación:

$$\frac{A}{B} \times C \times D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado

B = periodo de depreciación

C = Coste del equipo (Sin IVA)

D = % de uso que se dedica al proyecto

7.2.3 Coste del software

En este apartado se van a desarrollar los costes relacionados con el software necesario para el desarrollo del proyecto.

En el ordenador personal se encuentra instalado Windows XP, cuya licencia fue obtenida de forma gratuita a través de la universidad. Para la realización de esta memoria ha sido necesario utilizar Microsoft Office 2010 Hogar y Estudiante por el que hubo que abonar la licencia. Para la implementación del sistema ha sido necesaria la utilización del programa KEIL uVision, del cual fue suficiente con utilizar la versión de estudiante, que es gratuita.

Los costes nombrados aparecen detallados en Tabla 29.

Concepto	Coste (€)
Licencia Windows XP	0
Licencia Microsoft Office 2010 Hogar y Estudiante	89
Programa KEIL uVision	0
TOTAL	89 €

Tabla 29. Coste del software.

7.2.4 Coste del material de oficina

El coste del material de oficina que se ha utilizado para este proyecto se estima en aproximadamente 150 €.

7.2.5 Costes indirectos

Los costes indirectos son la partida destinada a gastos como pueden ser la luz, el agua, etc. Para este proyecto se asigna un 20% de costes indirectos sobre el coste total del proyecto.

7.2.6 Coste total

En este apartado se va a calcular el coste total del proyecto, apoyándonos en los costes obtenidos en los apartados anteriores. Esto se puede ver detallado en la figura 38.

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor: Eduardo Bonilla Palencia

2.- Departamento: Departamento de Tecnología Electrónica

3.- Descripción del Proyecto:

- Título: Implementación del algoritmo AES sobre arquitectura ARM con mejoras en rendimiento y seguridad
- Duración (meses): 6
- Tasa de costes Indirectos: 20%

4.- Presupuesto total del Proyecto (valores en Euros):

22.788,51 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Bonilla Palencia, Eduardo		Analista	2,224	5.250,00	11.676,00	
Bonilla Palencia, Eduardo		Programador	1,706	2.625,48	4.479,07	
Bonilla Palencia, Eduardo		Ingeniero Pruebas	0,228	3.937,50	897,75	
Mengibar Pozo, Luis		Ingeniero Senior	0,304	5.250,00	1.596,00	
					0,00	
		Hombres mes	4,462	Total	18.648,82	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Ordenador de sobremesa	700,00	100	6	60	70,00
Placa de desarrollo LPC 2132	20,00	100	6	60	2,00
ULINK2 JTAG Debugger	300,00	100	6	60	30,00
Micrófono electrect y alimentación	6,00	100	6	60	0,60
					0,00
					0,00
				Total	102,60

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} x C x D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
Licencia Windows XP		0,00
Licencia Microsoft Office 2010 Hoga		89,00
Programa KEIL uVision		0,00
Material Oficina		150,00
	Total	239,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,...

6.- Resumen de costes

Presupuesto Costes Totales	Presupuesto Costes Totales
Personal	18.649
Amortización	103
Subcontratación de tareas	0
Costes de funcionamiento	239
Costes Indirectos	3.798
Total	22.789

Figura 38. Presupuesto total del proyecto.

Se concluye finalmente que el presupuesto total de este proyecto asciende a la cantidad de **22.788,51€.**

Madrid a 3 de Mayo de 2012

El ingeniero proyectista

Fdo. Eduardo Bonilla Palencia

Glosario

ADC	*Analog to Digital Converter*
AES	*Advanced Encryption Standard*
ASIC	*Application-Specific Integrated Circuit*
ARM	*Advanced RISC Machines*
CPU	*Central Processing Unit*
DES	*Data Encryption Standard*
FPGA	*Field Programmable Gate Array*
GSM	*Groupe Spécial Mobile*
GF	*Galois Field*
JTAG	*Joint Test Action Group*
NIST	*National Institute of Standards and Technology*
USB	*Universal Serial Bus*

Referencias

1. **Joerg Eberspaecher, Hans-Joerg Voegel, Christian Bettstetter.** *GSM Switching, Services, and Protocols.* s.l. : John Wiley and Sons, 2001.

2. *Diccionario de la lengua española.* s.l. : Real Academia Española, 2001.

3. *A mathematical theory of communication.* **Shannon, Claude Elwood.** s.l. : Bell System Technical Journal, Julio y Octubre de 1948, Vol. 27, págs. 379-423 y 623-656.

4. *Communication Theory of Secrecy Systems.* **Shannon, Claude Edwood.** s.l. : Bell System Technical Journal, 1949, Vol. 28, págs. 656–715.

5. *New directions in cryptography.* **Diffie, Whitfield y Hellman, Martin.** s.l. : IEEE Transactions on Information Theory 22, 1976, Vol. 22, págs. 644-654.

6. **Feihu, Xu, Bing, Qi y Hoi-Kwong, Lo.** Experimental demonstration of phase-remapping attack in a practical quantum key distribution system. [En línea] 13 de Mayo de 2010. [Citado el: 30 de Enero de 2012.] http://arxiv.org/pdf/1005.2376.pdf.

7. **Nechvatal, James, y otros.** *Report on the Development of the Advanced Encryption Standard (AES).* s.l. : National Institute of Standards and Technology, 2000. Disponible en http://csrc.nist.gov/archive/aes/round2/r2report.pdf el 13/08/2011.

8. *AES2 Conference Attendee Feedback.* s.l. : National Institute of Standards and Technology (NIST), 1998.

9. Secure4Net. *AES Frequently Asked Questions.* [En línea] [Citado el: 18 de Septiembre de 2011.] http://www.image-in.co.il/HTML/SEC4NET/aes_faq.html.

10. The Advanced Encryption Standard (Rijndael). [En línea] [Citado el: 9 de Enero de 2012.] http://www.eng.tau.ac.il/~yash/crypto-netsec/rijndael.htm.

11. **(NIST), National Institute of Standards and Technology.** Federal Information Processing Standards Publication 197. *ADVANCED ENCRYPTION STANDARD (AES).* [En línea] [Citado el: 16 de Junio de 2011.] http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

12. **Labrada, Reinier Torres.** uControl. [En línea] [Citado el: 2 de septiembre de 2011.] http://www.ucontrol.com.ar/wiki/index.php?title=El_microcontrolador.

13. **Semiconductors, NXP.** Single-chip 16/32-bit microcontrollers; 32/64/128/256/512 kB ISP/IAP flash with 10-bit ADC and DAC. [En línea] [Citado el: 15 de Septiembre de 2010.] http://www.nxp.com/#/pip/pip=[pip=LPC2131_32_34_36_38]|pp=[t=pip,i=LPC2131_32_34_36_38].

14. UM10120. *LPC2131/2/4/6/8 User manual.* [En línea] NXP B.V., 2010. [Citado el: 15 de Septiembre de 2011.] http://www.nxp.com/documents/user_manual/UM10120.pdf.

15. **N.V, Koninklijke Philips Electronics.** LPC2131/2132/2138. *Single-chip 16/32-bit microcontrollers; 32/64/512 kB ISP/IAP.* [En línea] 2004 de Noviembre de 18.

http://www.nxp.com/acrobat_download2/expired_datasheets/LPC2131_2132_2138_1.pdf.

16. **Chip, FTDI.** TTL-232R TTL TO USB SERIAL CONVERTER RANGE OF CABLES Datasheet Version 2.02. [En línea] Future Technology Devices International Limited, 2010. [Citado el: 15 de Septiembre de 2011.] http://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_TTL-232R_CABLES.pdf.

17. **Dworkin, Morris.** Recommendation for Block Cipher Modes of Operation, Methods and Techniques. *National Institute of Standards and Technology.* [En línea] 2001. [Citado el: 5 de Mayo de 2011.] http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

18. **Daemen, Joan y Rijmen, Vincent.** The Rijndael Block Cipher. *AES Proposal.* [En línea] 3 de Septiembre de 1999. [Citado el: 26 de Mayo de 2011.] http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

19. **GmbH, ARM Ltd and ARM Germany.** ARM Technical Support. *ULINK: DRIVER.* [En línea] KEIL. [Citado el: 12 de 12 de 2011.] http://www.keil.com/support/docs/2937.htm.

20. **Bogdanov, Andrey, Khovratovich, Dmitry y Rechberger, Christian.** Biclique Cryptanalysis of the Full AES. [En línea] 31 de 8 de 2011. [Citado el: 30 de Enero de 2010.] http://eprint.iacr.org/2011/449.pdf.

21. UM10314. *LPC3130/31 User manual.* s.l. : NXP Semiconductors, 2009. págs. 330-333.

22. Pre para microfóno electret. [En línea] [Citado el: 7 de marzo de 2012.] http://www.pablin.com.ar/electron/circuito/audio/premic/index.htm.

23. **Mangard, Stefan, Oswald, Elisabeth y Popp, Thomas.** *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security).* s.l. : Springer, 2007. págs. 228-231. ISBN-10: 0387308571.

24. **Lucena López, Manuel José.** *Criptografía y Seguridad en Computadores.* Versión 4-0.8.1. s.l. : Universidad de Jaén, 2010.

25. Seguridad en internet. *Criptografía simétrica.* [En línea] [Citado el: 26 de 12 de 2010.] http://www.eumed.net/cursecon/ecoinet/seguridad/simetrica.htm.

26. **NIST.** AES Algorithm (Rijndael) Information. [En línea] [Citado el: 22 de Junio de 2011.] http://csrc.nist.gov/archive/aes/rijndael/wsdindex.html.

27. **Levy, Steven.** *Cripto: Cómo los informáticos libertarios vencieron al gobierno y salvaguardaron la intimidad.* Madrid : Alianza, 2002.

28. **Kahn, David.** *The Codebreakers.* New York : Macmillan, 1967.